A PROGRAMMING MODEL AND PROCESSOR ARCHITECTURE FOR HETEROGENEOUS MULTICORE COMPUTERS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Michael D. Linderman
February 2009

© Copyright by Michael D. Linderman 2009
All Rights Reserved

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Professor Teresa H. Meng) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Professor Mark Horowitz)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Professor Krishna V. Shenoy)

Approved for the University Committee on Graduate Studies.

Abstract

Heterogeneous multicore computers, those systems that integrate specialized accelerators into and alongside multicore general-purpose processors (GPPs), provide the scalable performance needed by computationally demanding information processing (informatics) applications. However, these systems often feature instruction sets and functionality that significantly differ from GPPs and for which there is often little or no sophisticated compiler support. Consequently developing applications for these systems is difficult and developer productivity is low.

This thesis presents Merge, a general-purpose programming model for heterogeneous multicore systems. The Merge programming model enables the programmer to leverage different processor-specific or application domain-specific toolchains to create software modules specialized for different hardware configurations; and provides language mechanisms to enable the automatic mapping of processor-agnostic applications to these processor-specific modules. The Merge programming model has been prototyped on a heterogeneous system consisting of an Intel Core 2 Duo CPU and an Intel X3000 integrated GPU, and a homogeneous 32-way SMP system. Speedups of 3.6×–8.5× using the X3000 and 5.2×–22× using the 32-way system were achieved for a set of informatics applications relative to their single-core C reference implementations.

The Merge programming model facilitates heterogeneity throughout the system, including within the processor microarchitecture itself. This thesis presents novel chip-multiprocessor (CMP) with a modular microarchitecture that supports the integration of non-trivial specialized accelerators into GPP cores. The accelerators function as "add-ons", maintaining backward compatibility with existing designs, while offering features from GPUs and other more specialized systems. The extended CMP is evaluated using execution-driven simulation; speedups of up to 2.4× were achieved relative to the base CMP architecture.

Contents

Abstract **iv**

1 Introduction **1**

1.1 A Programming Model for Heterogeneous Systems 3

1.2 Extending Legacy CMP Architectures 6

1.3 Thesis Contributions 7

1.4 Thesis Overview 8

2 Background and Related Work **9**

2.1 Heterogeneous Systems 9

2.1.1 Tightly-Coupled Heterogeneous Systems 10

2.1.2 Loosely-Coupled Heterogeneous Systems 11

2.2 Programming Models for Heterogeneous Systems 13

2.2.1 Direct Compilation 13

2.2.2 Library-based Programming 15

2.2.3 Library Metaprogramming 17

3 Merge Programming Model **27**

3.1 Core Characteristics of Merge 27

3.1.1 Encapsulating Specialized Code 28

3.1.2 Interchangeable Concurrent Functions 31

3.1.3 Function Bundling 33

3.2 Usage Models 36

3.3 Library Construction 37

3.3.1 Structuring Computations for Particular Heterogeneous Hardware 37

3.3.2 Assembling Hardware Resources for Particular Computations 38

4 Merge Compiler and Runtime **41**

4.1 Merge Prototype Design 41

4.1.1 Map-Reduce Parallel Pattern 43

4.1.2 Function Bundling 45

4.1.3 Compiler and Runtime 48

4.2 Evaluation 51

4.2.1 Prototype Implementation 51

4.2.2 Evaluation Platforms 53

4.2.3 Performance Analysis 53

5 Heterogeneous Microarchitectural Extensions **59**

5.1 Modular Microarchitectures 60

5.1.1 Extension Interfaces 60

5.1.2 Extender Unit 61

5.2 Extension Units 62

5.2.1 Fetch Unit 63

5.2.2 Ring Network 65

5.2.3 Thread Interaction Unit 65

5.3 Programming Model 66

5.3.1 Execution Model 67

5.3.2 Integrating Extension-specific Code 68

5.3.3 Backward/inter compatibility 69

5.4 Evaluation 70

5.4.1 Experimental System 70

5.4.2 Comparing Base and Extended CMPs 71

5.4.3 Performance Implications of Backward/Inter Compatibility 75

5.4.4 Invasive Microarchitectural Changes 77

6 Conclusion **79**

Bibliography **81**

List of Tables

2.1 Summary of features of library metaprogramming tables 20

4.1 Map-reduce language syntax 44
4.2 Merge bundling and predicate-dispatch syntax 47
4.3 Benchmark Summary 52

5.1 Simulation parameters 71
5.2 Benchmark Summary 72

List of Figures

1.1 Computational complexity of informatics kernels 2
1.2 Commodity heterogeneous system performance 3
1.3 Sketch of Merge programming model 5

2.1 GPGPU programming models 14
2.2 Example usage of VSIPL++ library template dispatch system 17
2.3 Library metaprogramming tools 18
2.4 Typical architecture of library metaprogramming tools 19
2.5 Example metaprogramming annotation for Broadway Compiler 22
2.6 Dense matrix multiplication in Sequoia 24

3.1 Encapsulation of specialized languages 28
3.2 Sketch of EXOCHI framework 29
3.3 CHI compilation flow 29
3.4 CHI code example 30
3.5 Sketch of communication between accelerator(s) and CPU(s) 32
3.6 Sketch of function bundling 34
3.7 Function that might be reimplemented to improve performance while maintaining interchangeability 38
3.8 Assembling accelerators in resource sets 40

4.1 Decomposition of k-means algorithm 42
4.2 Sketch of Merge operation 43
4.3 Merge implementation of portion of k-means algorithm 46
4.4 Runtime variant selection and invocation flowchart 49
4.5 Merge compilation flow. 51

4.6 Speedup on heterogeneous system 54
4.7 Cooperative heterogeneous multi-threading scenarios 55
4.8 Speedup for different cooperative heterogeneous multi-threading scenarios 55
4.9 Speedup on heterogeneous system 57

5.1 Block diagram of extended CMP architecture 60
5.2 Block diagram of Fetch extension 63
5.3 Example fetch descriptors 64
5.4 Block diagram of the Thread Interaction Unit extension 66
5.5 Compilation flow 67
5.6 Sample code for histogram 67
5.7 Benchmark execution time on base and extended CMPs 73
5.8 Benchmark execution time on base and extended CMPs (cont.) 74
5.9 Amount and distribution of instructions executed 75
5.10 Execution time and distribution for emulated extensions 76
5.11 Execution time for direct register mapping 78

Chapter 1

Introduction

Heterogeneous multicore computers, those systems that integrate specialized accelerators into and alongside multicore general-purpose processors (GPPs), can provide the scalable performance that computationally demanding information processing (informatics) applications need and GPPs alone cannot deliver. These systems, however, often feature functionality that significantly differs from GPPs, and for which there is often limited or no sophisticated compiler support. Consequently developing applications for these systems is challenging and programmer productivity is low. There is an urgent need to assist the creators of these informatics workloads in taking advantage of state-of-the-art heterogeneous multicore computers.

The How Much Information [41] project estimates that in 2002 approximately 5 *exabytes* of new digital information was stored worldwide. The 2002 figures represented a $2\times$ increase over their previous estimate in 1999, suggesting a 30% annual growth rate. The size of these new datasets presents a significant challenge to computer system architects. Figure 1.1 shows the computational complexity for representative informatics applications as dataset sizes increase. For many applications, the dataset does not need to grow very large before the computation requires days or even weeks to complete using current state-of-the-art commodity GPPs.

Computers with sufficient computational performance can be created by assembling multiple GPPs in a single cluster, an approach used by most modern supercomputers. Since each core is relatively low performance, less than 100 gigaflop/s for commodity GPPs in 2008 [23], many processors are required. Each additional processor increases system cost, and reduces efficiency as measured by performance/watt, performance/dollar, and performance/volume [30]. For many embedded informatics applications, such biological implants or autonomous vehicles, the size, cost and power consumption of clusters of GPPs are prohibitive. And even domains, such as desktop

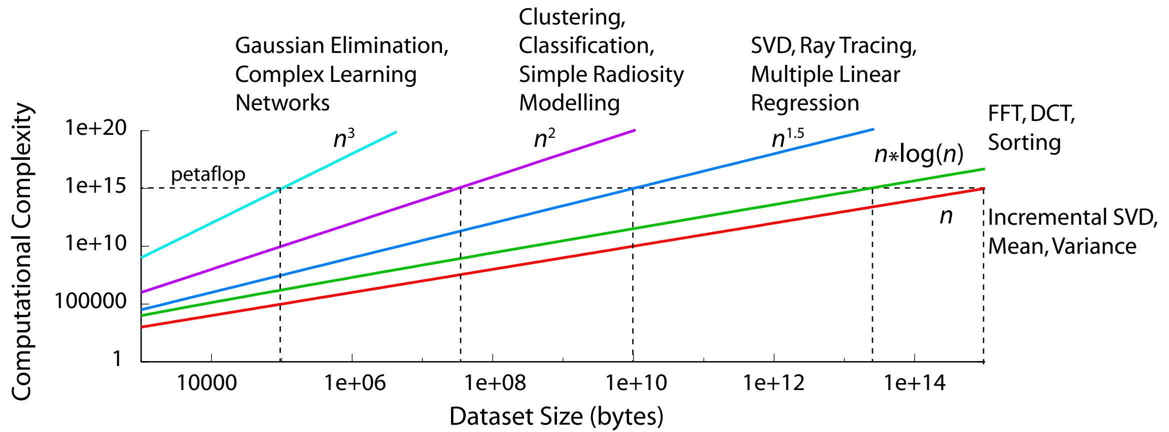


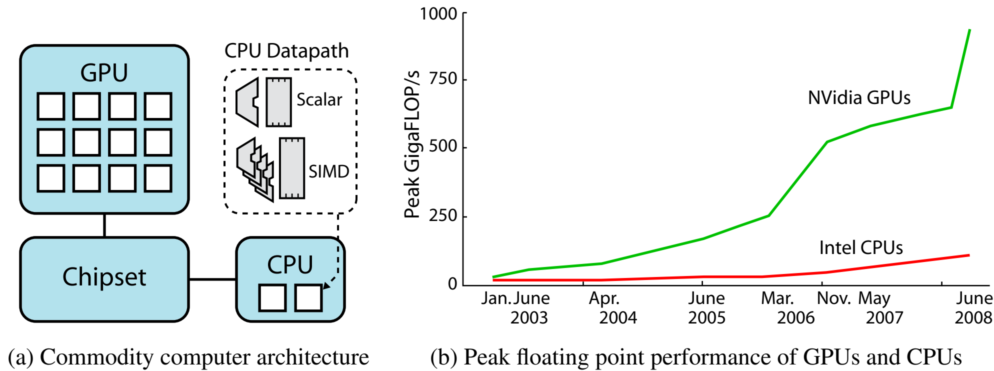
Figure 1.1: Computational complexity as a function of dataset size for different informatics algorithms. Dotted line indicates workload sizes requiring a petaflop. Adapted from [17].

computing and servers, where efficiency has not traditionally been a concern are starting to become sensitive to energy and volume efficiency [7].

Unfortunately, GPP performance scaling trends suggest that the gap between the computational requirements of informatics applications and the capability of GPPs will continue grow. Historically, the performance of single-core GPPs has improved about 52% per year (1986-2002); however, starting in 2002 the rate of improvement has slowed to about 20% [30]. At the present rate, single-core GPP performance is now scaling slower than the workloads are growing.

Significant improvements in computer system performance and efficiency are required. Fortunately, more capable heterogeneous multicore computers are becoming widely available. Figure 1.2a shows a block diagram of a typical commodity computer system, which includes both tightly-coupled (SIMD functional units) and loosely-coupled (graphics processing unit, or GPU) accelerators. The heterogeneous systems can deliver orders-of-magnitude improvement in performance and energy efficiency relative to their GPP-only counterparts. For example, Figure 1.2b shows the peak floating-point performance for several generations of GPPs and GPUs. For approximately the same dollar, energy and volume cost, GPUs deliver both better performance in absolute terms and better performance scaling (GPU peak performance is scaling greater than 100% per year).

GPUs [53] and SIMD extensions, such as SSE extensions to x86 [50, 56], are the best known and most widely available examples of a much larger heterogeneous system design space. System designers are also incorporating reconfigurable fabrics [29], such as discrete field programmable



(a) Commodity computer architecture (b) Peak floating point performance of GPUs and CPUs

Figure 1.2: Construction (a) and peak floating point performance (b) of commodity heterogeneous systems. Includes both loosely-coupled (GPU) and tightly-coupled (SIMD functional unit) heterogeneous accelerators. Performance data adapted from [49].

gate arrays (FPGAs) and reconfigurable functional units, domain accelerators, such as cryptographic units [68], and other fixed-function accelerators into and alongside GPPs to improve performance and efficiency.

Heterogeneous systems improve performance and efficiency by exposing to programmers architectural features, such as low latency software-controlled memories and inter-processor interconnect, that are either hidden or unavailable in GPPs. The software that executes on these accelerators often bears little resemblance to its CPU counterpart: source languages and assembly differ, and often entirely different algorithms are needed to take advantage of the capabilities of the different hardware.

This thesis takes a two-pronged approach to enabling informaticians to exploit complex heterogeneous systems: (1) provide a programming model capable of integrating the different compiler toolchains and algorithmic approaches needed by diverse accelerators; and (2) incorporate key architectural features from accelerators into current chip-multiprocessor architectures (CMPs) to improve performance and efficiency while maintaining backward compatibility and minimizing the barriers for adoption.

1.1 A Programming Model for Heterogeneous Systems

A diverse set of vendor-specific, architecture-specific and application-specific programming models continue to be developed to exploit these new and different hardware resources. Most of these

tools, however, focus on supporting general-purpose computation on a particular accelerator, such as GPUs, or class of accelerators, such as SIMD and SPMD architectures (of which GPUs are a member). However, as Figure 1.2a shows, specialized accelerators, such as GPUs, are part of a larger *system*. At a minimum, the GPU is integrated alongside a single-core GPP with SIMD extensions, and much more likely, integrated alongside a multicore GPP and other heterogeneous processors such as FPGAs [29], or the Cell processor [31]. Achieving system performance and energy efficiency goals requires smartly distributing the computation across all of the available hardware resources.

Programming language-based efforts, such as streaming [9, 63, 35, 69, 43] and matrix-oriented languages [60, 44, 54], focus on compiler-driven optimizations for these execution models for particular classes of accelerators. Adapting traditional compiler techniques, however, binds these tools too closely to their choice of target architectures. The canonical compiler reduces a computation expressed in a high-level language to a small, fixed set of primitive operations that abstract the capabilities of the target hardware. The choice of primitives biases all aspects of the compiler, including the source languages that can be supported, the optimizations that can be applied and the hardware that can be targeted. As such, capabilities developed for one set of primitives can be of limited use when those primitives fail to abstract important aspects of a different application domain or machine.

Unfortunately, there is no one set of primitives that can be used to abstract all of the unique and specialized capabilities provided by different hardware. Scalar processors are best abstracted with scalar three-address operations, SIMD processors (*e.g.* GPUs, the Cell processor, SSE units) are better abstracted with short-vector operations, while FPGAs are better abstracted with binary-decision diagrams and data-flow graphs with variable-precision operands. Much as the limitations of scalar primitives motivated the adoption of short-vector primitives in tools targeting SIMD architectures, compilers that target complex heterogeneous systems with specialized accelerators, such as FPGAs, will find representations based on simple scalar or short-vector primitives limiting and ineffective. To target diverse and evolving heterogeneous systems, variable and inclusive primitives, primitives that can abstract computational features of any complexity (variable) and for any architecture, or using any programming model (inclusive), are needed.

This thesis proposes *Merge*, a general-purpose programming model for complex heterogeneous systems. Merge is motivated by the realization that programming languages already provide a type of variable and inclusive primitive – functions – that can serve as the basis for programming system that readily integrates different programming models. Merge, summarized in Figure 1.3, replaces

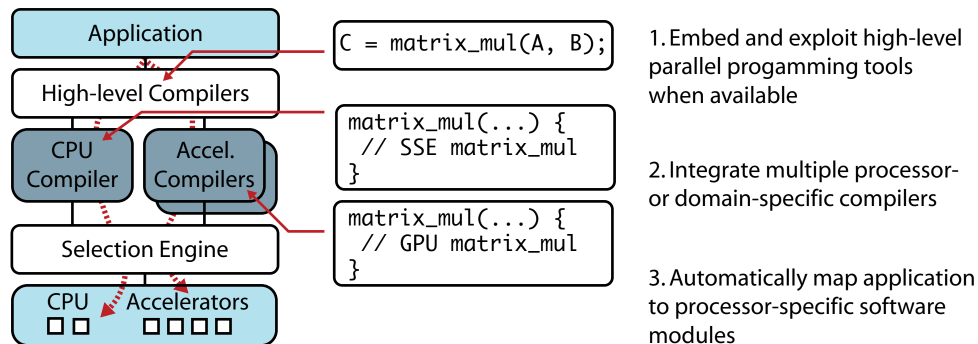


Figure 1.3: Sketch of Merge programming model

current ad hoc approaches to programming for heterogeneous platforms with a rigorous, library-based methodology that automatically maps processor-agnostic applications to specialized functions that are implemented with different processor-specific or domain-specific toolchains.

Merge uses EXOCHI [64] and other tools to *encapsulate* domain-specific languages or accelerator-specific languages in C/C++ functions to provide a uniform interface and inclusive abstraction. The resulting C/C++ code is largely indistinguishable from existing ISA intrinsics, such as those targeting SSE extensions, enabling the programmer to create new intrinsics of any complexity, for any architecture and using any programming model supported by the encapsulation tools. Different implementations of a function are *bundled* together, creating a layer of indirection between the caller and the actual implementation that facilitates the mapping between application and implementation.

Merge is effectively a form of library metaprogramming; it attempts to improve performance by optimizing across function calls (and function implementations) using programmer supplied annotations. The Merge compiler is not responsible for directly generating machine code for a particular platform. Instead it is a tool for integrating different processor-specific or application domain-specific toolchains. Merge uses the limited scope and defined interface of C/C++ functions to define quanta of computation that can potentially be mapped to different hardware resources and otherwise optimized by the compiler.

From the perspective of the programmer, Merge exposes a range of granularities at which *he* or *she* can substitute new and specialized implementations for those provided for by existing compiler technology. Relying exclusively on the code generation capabilities of existing compilers, even those that offer compiler intrinsics, fixes the granularity of abstraction (as discussed above), and effectively excludes the programmer from incorporating their own specialized implementations. A purely ad-hoc approach, while extremely flexible, often results in applications that are difficult to

extend to take advantage of new or different hardware resources. Merge offers a middle ground between these two extremes, exploiting the flexibility of functions as primitives, while providing tool support for managing these functions to ensure maintainability and extensibility.

1.2 Extending Legacy CMP Architectures

In contrast to specialized accelerators, such as GPUs, that have changed significantly over the past 20 years, the ISAs of commodity GPPs have changed remarkably little. More recent additions, such as SIMD extensions, have introduced more complex arithmetic units, but made no changes to the basic execution semantics, the memory system, or how different processor cores interact. The performance of GPUs, the Cell processor, and other specialized or heterogeneous systems, however, shows the potential benefit of integrating software-controlled local memories into the memory hierarchy, and providing low-latency interconnect and atomic interaction facilities for inter-processor interaction.

Unfortunately, at present, such features are typically only available in discrete accelerators, such as GPUs, that are loosely-coupled to the host GPP. The loose coupling can introduce significant data and control-transfer overheads, particularly for applications that require fine-grain cooperation between the GPP and the accelerator. Integrating key architectural features from discrete accelerators into existing GPP microarchitectures as optional extensions can bring the capabilities of specialized accelerators to GPPs, but with reduced overhead and barriers to entry.

This thesis proposes a novel extensible chip-multiprocessor (CMP) architecture, motivated by our experiences with the Merge programming model, that can integrate non-trivial extensions, such as software-controlled data fetch units, that improve performance and efficiency, while maintaining backward compatibility with existing GPP architectures. Three extensions are presented: a per-core data fetch engine, a thread interaction unit, and a ring-based communication network. The Merge programming model is used to encapsulate domains-specific languages for each extensions, provide code selection for backward or inter-compatibility to non- or differently-extended architecture, and facilitate flyweight cooperation between the processor core and the concurrently executing extensions.

The extension units' interface, the extender (EXT), is integrated into the processor's load/store (LS) unit and can support a range of different extensions, or combinations of extensions, without any changes to the base core's datapath or instruction decoder. Memory accesses are de-multiplexed in the EXT unit to either the memory system or one of the extension units. Interaction with the

extensions can thus be interleaved with standard instructions on a per-instruction basis, offering flyweight cooperation between the CPU core(s) and the extension units. The extensions are not just new functional units, however; each command can launch an asynchronous, concurrently executing thread on the extension, making these units more analogous to application-level sequencers [28]. The use of the extensions is optional, making extended systems backward compatible with legacy code. In contrast to system-level accelerators, in which use of specialized functionality is an all-or-none proposition, the programmer can selectively take advantage of the extended CMP's new features.

1.3 Thesis Contributions

This thesis makes the following contributions:

- I describe Merge, a pragmatic library-based programming model for developing applications for heterogeneous systems. Merge enables programmers to leverage different processor-specific or domain-specific toolchains to create software modules specialized for different hardware configurations, and provides language mechanisms to enable the automatic and dynamic mapping of the application to these processor specific modules.

- I present a prototype implementation of the Merge compiler and runtime, and report significant performance improvements for information processing applications on heterogeneous (Intel Core 2 Duo processor and Intel X3000 GPU) and homogeneous (Unisys 32-way SMP with Intel Xeon processors) platforms relative to the C reference implementation.

- I present a modular microarchitecture that supports the addition of tightly-coupled asynchronous extensions to legacy CMPs. The extended CMPs bridge the gap between loosely-coupled discrete accelerators, such as GPUs, and tightly-coupled extensions, such as SSE instructions. I describe how the Merge programming model can be used to develop applications for extended CMPs.

- I describe three extension modules, and provide a detailed evaluation of an extended CMP using an execution-based simulator. I report significant speedups relative to the base CMP for information processing applications.

1.4 Thesis Overview

This thesis describes the Merge programming model and a heterogeneous microarchitectures motivated by our experiences working with Merge. The thesis begins with an overview of current heterogeneous systems and the programming models developed for these platforms in Chapter 2. The discussion of related work in Section 2.2 focuses on current library metaprogramming tools, the existing programming models most similar to Merge.

Chapter 3 introduces the Merge programming model. Section 3.1 describes the three key characteristics and capabilities of Merge, encapsulation, interchangeability, and bundling, and how they address the challenges introduced in Chapter 2. Sections 3.2 and 3.3 show how encapsulation and bundling can be used to manage resources in complex heterogeneous systems, enable aggressive compiler optimizations, and support different scheduling techniques.

Chapter 4 describes the prototype implementation of the Merge programming model. Section 4.1 describes the design of the prototype, focusing on the encapsulated map-reduce-based parallel programming language incorporated into the prototype compiler and runtime. Section 4.2 provides a detailed evaluation of the prototype on both a heterogeneous system consisting of Intel Core 2 Duo processor and an Intel X3000 GPU, and a homogeneous 32-way SMP system.

Chapter 5 describes the extensible CMP architecture. Sections 5.1 and 5.2 present the modular microarchitectures and the three extension units; Section 5.3 presents the programming model, based on Merge, for the extended CMP; and Section 5.4 provides a detailed performance evaluation.

Chapter 6 concludes the thesis and provides a roadmap for future research on both the Merge programming model and extensible CMP architectures.

This section is fine, but for the future I think the real point of this section is NOT to say what the thesis (paper) is about, but rather why is the material in the paper – it should motivate what follows so the reader knows why you are going to tell him/her this stuff.

Chapter 2

Background and Related Work

Exponentially more demanding information processing (informatics) workloads are driving a corresponding increase in the performance and efficiency requirements for computer systems. Heterogeneous systems, which integrate specialized accelerators alongside, or into, general-purpose processors (GPPs), can deliver significant improvements in performance and efficiency. This chapter provides background on heterogeneous systems, and discusses in detail the different programming models that have been proposed to develop applications for these systems.

2.1 Heterogeneous Systems

Heterogeneous Systems encompass a broad set of system and processor architectures. These systems integrate computational resources that feature instruction sets and functionality that significantly differ from general-purpose processors (GPPs). These systems typically fall into one of two categories, classified by their execution model: (1) an ISA-based tightly-coupled model, or (2) a device driver-based loosely-coupled execution model [64]. Prominent examples of the tightly-coupled approach are the SIMD SSE extensions to the x86 ISA [50, 56]; common examples of the loosely-coupled model are systems that use the graphics processing unit (GPU) for general-purpose computing (termed GPGPU) [53].

As described in Chapter 1, the interest in heterogeneous systems, both tightly-coupled and loosely-coupled, is motivated by increasingly demanding performance and efficiency requirements. Specialized accelerators, such as GPUs, are some of the most powerful and most efficient commodity processors available. State-of-the-art GPUs in 2008 deliver approximately 10× better peak floating point performance than commodity GPPs for approximately the same dollar cost and with

similar power consumption (about 1 teraflop/s peak for the GPU vs. about 100 gigaflop/s for the multicore GPP[49]). Further, the performance of specialized accelerators is increasing at a faster rate; measured with graphics performance metrics, GPU performance has improved yearly by 1.7× (pixels/second) to 2.3× (vertices/second) [53]. This rate of improvement significantly outpaces single-core GPPs, which are improving 1.2× – 1.5× per year [30] (measured with the SPEC benchmark suites).

Accelerator performance is improving at a faster rate than that of GPPs because accelerators devote more transistors to arithmetic units and other functional blocks that actually perform the computation. Accelerators and GPPs benefit equally from advances in semiconductor fabrication technology. However, GPPs, devote many transistors to caches, branch-predictors, instruction windows and other structures designed to extract instruction-level parallelism from sequential applications. The significant data and task-level parallelism in graphics computations, and other accelerated workloads, enables GPUs and other accelerators to use transistors directly for computation, and thereby achieve increased arithmetic performance relative to GPPs with the same number of transistors [53].

Accelerators are able to use transistors more efficiently because they are specialized for a particular computation or class of computations. Outside that restricted domain, the accelerator can be of limited or no use. GPUs, for instance, have limited or no support for double precision floating point arithmetic; and a cryptographic accelerator [68] will be limited to common encryption and decryption algorithms. Heterogeneous *systems*, which integrate specialized accelerators and GPPs, attempt to exploit the performance and efficiency of the accelerator while maintaining the performance of legacy applications and other codebases that cannot take advantage of the accelerators' unique or different capabilities.

The two categories of heterogeneous systems, tightly-coupled and loosely-coupled, primarily differ in which types of computations they seek to accelerate, and how and at what granularity the accelerators will interact with the GPP. The following sections describe both categories in more detail.

2.1.1 Tightly-Coupled Heterogeneous Systems

Tightly-coupled heterogeneous accelerators are logically integrated into the CPU datapath, and are typically targeted with customized instructions that have been added to the ISA. The accelerators are treated as alternate functional units, and are commonly designed to accelerate complex arithmetic operations, such as linear interpolation or short-vector SIMD arithmetic, that might otherwise need

to be implemented as a sequence of CPU instructions.

The x87 floating-point operations [21], SIMD extensions like the SSE instructions cited previously, MIPS coprocessor instructions [51], reconfigurable functional units [29], Tensilica's TIE instruction extensions [24] and other application-specific instruction processors (ASIPs) [10, 19] are examples of tightly-coupled accelerators.

Tightly-coupled accelerators typically include specialized register files that are integrated into the microarchitecture alongside the existing CPU registers. The SSE extensions, for example, add a set of 128-bit short-vector registers, named `xmm0 -- xmm15`, to the x86 ISA architectural registers. Data can be moved between the CPU and the accelerator registers at word granularity, enabling low-overhead, fine-grain cooperation between the CPU and the accelerator. Accelerator instructions are decoded and issued by the CPU, enabling accelerator and CPU operations to be interleaved at the granularity of individual instructions, further facilitating low-overhead cooperation.

The tight-coupling can, however, limit the types of operations that can be accelerated. Accelerator operations must be compatible with the CPU execution model, restricting, for example, the amount and type of data that can be accessed by the accelerator. Tensilica TIE extensions, for instance, are limited to loading 32, 64 or 128 bits from memory in a single operation. Since the accelerator executes synchronously with the CPU instruction stream, the execution of the accelerator cannot be decoupled from the CPU to hide communication, computation, or memory access latencies.

2.1.2 Loosely-Coupled Heterogeneous Systems

Loosely-coupled heterogeneous accelerators are integrated alongside GPPs using an on-chip or off-chip interconnect. Although the accelerator hardware might be integrated on the same silicon die as the CPU, the two processors are logically distinct. Interaction typically takes the form of bulk memory and command transfers between the CPU's and accelerator's unique and separate memory spaces. GPUs [53], discrete FPGAs and other reconfigurable fabrics [29], the CELL processor's synergistic processing elements (SPEs) [31], cryptographic [68], TCP offload and other application-specific accelerators are examples of loosely-coupled accelerators.

Compared to tightly-coupled accelerators, the granularity of interaction between the GPP and the loosely-coupled accelerator is much coarser. The latency and bandwidth limitations of these coarse grain connections (e.g., system buses, memory controllers) can introduce significant data and control-transfer overhead, especially for applications that require fine-grain cooperation between the CPU and the accelerator. For example, data transfers between the CPU and PCI-E connected

NVIDIA GPUs can sustain bandwidths of 1 – 3 GB/s, while the sustainable memory bandwidth for both the GPU and the CPU are considerably larger (10+ GB/s for the state-of-the-art x86 CPUs, and 50+ GB/s for state-of-the-art GPUs in 2008) [53]. A benefit of this coarse integration, however, is much more flexibility for the accelerator architect. The design of tightly-coupled accelerators is often limited by the execution semantics of the GPP, while the design of loosely-coupled accelerators is not. As a result, loosely-coupled accelerators can offer radically different instruction semantics, memory systems and resources for inter-processor interaction than their GPP counterparts.

The experimental system used in thesis is built around a commodity GPU, the Intel X3000 chipset-integrated GPU [14]. The X3000 GPU provides a single-program multiple-data (SPMD) execution model. The X3000 contains eight programmable execution units (EUs). Each EU includes a 256-bit wide predicated SIMD functional unit that can execute up to 8 single-precision floating operations in parallel, has a large local register file (in excess of 8 kB), and supports four hardware thread contexts for simultaneous multithreading (SMT). Each EU has a simple scalar in-order pipeline, with no branch prediction or other support for speculative instruction issue. Pipeline and memory access latencies are hidden by the four-thread hardware SMT. The combination of a simple pipeline, SIMD functional units, and SW-controlled memories (instead of caches) improves energy efficiency by approximately 30× relative to an Intel Core 2 Duo processor [66, 14]. The tradeoff, however, is that applications targeting the X3000 must be structured differently compared to their GPP-only counterparts.

To efficiently use the graphics fabric, the programmer must be able to create many SPMD microthreads (called shaders in graphics terminology) that can be dispatched by the GPU's hardware thread scheduler. Each microthread must load or store data from main memory in the large register file using 256-byte bulk transfers, and must carefully parallelize the computation across the SIMD execution lanes. Unlike applications written for GPPs, in which sound general programming practice will likely yield satisfactory performance across many different processors, programs that effectively exploit the X3000 are specifically optimized for its execution model, and in particular the size of its register file and other SW-controlled resources. Applications that are not specifically tailored for the X3000 will either be unusable or perform poorly. One too many threads (*e.g.* 33 threads, when the X3000 can concurrently execute 32 thread contexts) or a change in vector length (*e.g.* 8 to 12) will result in an increase in execution time that is disproportionate to the increase in workload size. Thus applications that target specialized accelerators will not only need to be structured differently than those targeting GPPs, but will also need to be structured differently for other accelerators, or even different generations of the same accelerator.

2.2 Programming Models for Heterogeneous Systems

A diverse set of vendor-specific, architecture-specific and application domain-specific programming models have and are currently being developed to exploit the new and different hardware resources available in heterogeneous systems. The following sections describe three different categories of programming models for heterogeneous systems: (1) direct compilation, (2) library-based programming, and (3) library metaprogramming.

Direct compilation is the programming language-based approach to raising the level of abstraction at which the programmer interacts with heterogeneous hardware. However, for the reasons described in Chapter 1 and in further detail in the following sections, direct compilation tools can be ineffective for complex heterogeneous systems that include multiple different processors, each of which might be best targeted with a different programming model.

Library-based programming is an alternate approach to raising the level of abstraction. Encapsulating implementations targeting specialized accelerators in software libraries hides the complexity of the underlying code, enabling non-expert programmers, using existing languages, to exploit heterogeneous systems. However, applications that make extensive use of libraries, often become tightly bound to their choice of library, and functions within that library, potentially making it difficult to selectively change implementations to target new architectures.

Library metaprogramming tools introduce a layer of indirection between the function call and the function implementation that can be used to address the above limitations and facilitate optimization across function calls (and function implementations). Heterogeneous systems, however, introduce additional challenges and requirements, such as supporting accelerator-specific programming models, ensuring extensibility for new or different hardware, and enabling dynamic specialization for shared hardware resources (*e.g.* GPU). These features were less relevant for metaprogramming tools designed for homogeneous GPPs, and so often have limited or no support in existing library metaprogramming systems.

2.2.1 Direct Compilation

Direct compilation tools directly translate the source language to the accelerator ISA or API. As with GPPs, a range of abstractions are available to the programmer, from accelerator-specific assembly to high-level parallel languages and runtimes. To focus the discussion, this thesis will almost exclusively use GPGPU examples (summarized in Figure 2.1) to demonstrate the properties of different types of direct compilation programming models for heterogeneous accelerators.

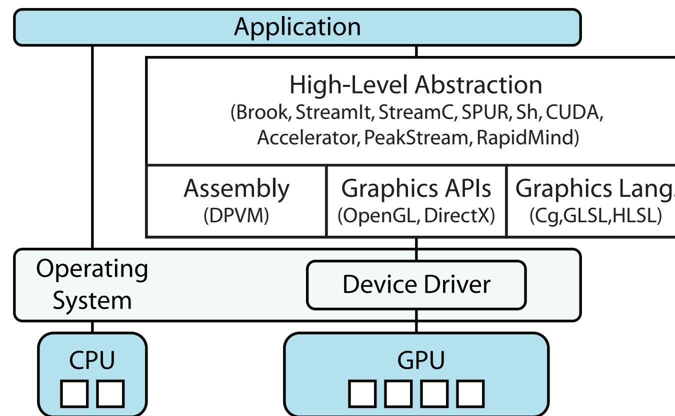


Figure 2.1: Summary of GPGPU programming models at different level of abstraction. Adapted from [64].

GPUs have been adapted for general-purpose use from their existing role in 3-D graphics acceleration; the low-level programming models often reflect their original purpose. At the lowest-level of abstraction, the Data Parallel Virtual Machine [58] directly exposes GPU assembly to programmer. Low-level vendor-specific tools (such as CUDA [47] and CTM [58]) expose the GPU architecture almost as directly, although with a more C-like syntax. Domain-specific graphics APIs (such as OpenGL and DirectX), along with graphics shader languages (such OpenGL Shader Language, Cg [42], and HLSL), provide a higher-level abstraction for the GPU, hiding the ISA and any vendor or device-specific implementation details. However, the APIs and programming languages designed for graphics applications can be difficult to use for algorithms outside that domain. For example, the programmer is required to map the application to graphics datatypes (vertex, pixel) and operations (render-to-texture) that are unwieldy for many general-purpose computations.

Higher-level programming models offer more general, although still application domain-specific, abstractions for the GPU. Streaming programming models, such as Brook [9], StreamIt [63], Sh [43], StreamC [35], and SPUR [69], abstract the GPU as a stream processor. The programmer writes small kernels using familiar C-like syntax, that can be applied in parallel to data objects drawn from a stream of such objects. Matrix-oriented domain-specific virtual machines (DSVMs), such as Accelerator [60], RapidMind [44] and PeakStream [54], compile high-level data-parallel languages to a GPU's ISA or a graphics API. These approaches provide a data-parallel API that can be mapped to the SIMD and SPMD functionality provided by GPUs and other accelerators, (*e.g.*, the Cell SPEs and x86 SSE extensions).

When an application fits the particular application domain, be it graphics, streams or matrix

computations, the programming environments cited above can improve programmer productivity and application performance. The restricted semantics of streaming languages, for instance, enable sophisticated algebraic optimizations normally performed manually by digital signal processing (DSP) engineers [1]. Outside of those domains, however, these tools can be of limited or no use. For example, while the matrix primitives used by RapidMind and the other DSVMs are appropriate for SIMD and SPMD functionality provided by GPUs, SSE extensions and the Cell processor, they will be inadequate and ineffective for representing more complex MIMD functionality offered by many heterogeneous systems.

Specialized accelerators, particularly loosely-coupled accelerators, are just one part of a larger heterogeneous system. For example, at a minimum, a GPU is integrated alongside a single-core GPP, and more likely is one of several accelerators alongside a multi-core GPP. As such, programming models need to be designed to not just support general-purpose computation on a particular specialized accelerator, but to exploit the entire heterogeneous *system* by smartly distributing work across all the available processors. The tools cited above only distribute work across different processors in very limited contexts, such as among the Cell SPEs or homogeneous multicore GPPs. In most cases the programmer must distribute the application manually across different processors.

Some tools are more amenable to manually structuring applications than others. The graphics APIs and the matrix-oriented DSVMs are both designed to be integrated into C/C++ applications. In contrast most of the streaming languages are designed to be stand-alone programming environments. Tools compatible with a more traditional C/C++ environment can better leverage existing software infrastructure, and potentially be integrated alongside other similar tools designed for other accelerator architectures or application domains.

These issues are not exclusive to programming models targeting GPUs; direct compilation tools targeting other accelerators, such as FPGAs, have similar tradeoffs.

2.2.2 Library-based Programming

High-level programming models are designed to improve programmer productivity by raising the level of abstraction. Similar improvements in productivity can also be achieved with expert-created libraries that implement common computations on specialized accelerators. The BLAS linear algebra libraries that have been released for the Cell processor [32] and NVIDIA GPUs [48] are examples of this approach. Building a specialized implementation of a computation, *e.g.* matrix multiply, for a particular accelerator is much simpler than building a compiler, making specialized accelerators available to non-experts sooner, and with a smaller initial investment in software

infrastructure.

The primary tradeoff for libraries is limited scope: only some computations will be ported to a particular accelerator. However, for state-of-the-art accelerators with limited or no sophisticated compiler support, a library-based programming might be the only way to make specialized hardware resources accessible to non-experts. When suitable libraries are available, the challenge for the programmer is ensuring that the particular implementation is effective and efficient across the range of possible input and machine configurations. Applications are tightly-bound, however, to their choice of libraries, making it difficult to selectively integrate new implementations that are better optimized for particular input or machine configurations.

Programmers typically must build function selection into their application using a combination of static and dynamic techniques such as `#idef` statements, static/dynamic linking and `if-else` blocks. The C standard library, for instance, is specialized through system-specific linking. However, as the diversity of heterogeneous systems increases and systems with multiple accelerators become commonplace, the number of implementations available will make such approaches impractical. Users often need to keep track of all the different function implementations that are available (termed variants), create manual dispatch code, and maintain multiple platform-specific versions of each program, all of which make applications non-portable and difficult to maintain.

Some libraries incorporate boolean predicates to automatically control function specialization. The VSIPL++ signal processing library [13] uses C++ template metaprogramming to provide both static and dynamic guards for each function implementation. An example usage of the VSIPL++ template dispatch system is shown in Figure 2.2. The C++ template system is used to create a dispatch wrapper, shown in Figure 2.2b, that can be called in place of any particular implementation; the wrapper traverses a list of available implementations, invoking the first implementation whose predicate function evaluates to `true`. The dispatch wrappers create a level of indirection between the function call and the actual implementation that enables the VSIPL++ authors to easily incorporate new implementations that are optimized for particular input or machine configurations. This addresses some of the limitations of library-based programming identified above.

In effect, the VSIPL++ library implements a very simple form of library metaprogramming. To maintain compatibility with existing C++ compilers, however, the VSIPL++ developers restrict themselves to the metaprogramming capabilities of the C++ template system. As a result they are limited in the optimizations they can perform, such as reordering the function variants to ensure better performing variants are preferentially selected. The function metaprogramming tools described

```
// Type name to tag operation
struct Op;

// A and B variants for Op
template <>
struct Evaluator<Op, A>
{
  static bool const ct_valid = true;
  static bool rt_valid(int &result, int i) { return i < 10;}
  static void exec(int &result, int i) { result = 0;}
};
template <>
struct Evaluator<Op, B>
{
  static bool const ct_valid = true;
  static bool rt_valid(int &result, int i) { return i >= 10;}
  static void exec(int &result, int i) { result = 1;}
};
```

(a) Source implementation using VSIPL++ dispatch system

```
void dispatch<Op, void, int& int>(int& result, int i) {
  if (Evaluator<Op, A>::ct_valid && Evaluator<Op, A>::rt_valid(r, i))
    Evaluator<Op, A>::eval(r, i);
  else if (Evaluator<Op, B>::ct_valid && Evaluator<Op, B>::rt_valid(r, i))
    Evaluator<Op, B>::eval(r, i);
  else
    // Raise error
}
```

(b) Output of template instantiation for VSIPL++ dispatch system

Figure 2.2: Example usage of VSIPL++ library template dispatch system for computation `Op`, with variants, `A` and `B`, each with compile time, `ct_valid`, and runtime, `rt_valid`, guards

in the next section integrate support for library metaprogramming into the compiler to enable optimization across function implementations in ways that VSIPL++ cannot.

2.2.3 Library Metaprogramming

Library metaprogramming techniques provide language support to address the portability and maintainability challenges that arise with traditional library-based programming for heterogeneous systems. These tools expose the library semantics directly to the programmer to improve performance by optimizing across function calls. Like library-based programming, specialized implementations are encapsulated within functions to raise the level of abstraction. However, instead of being an opaque encapsulation, each function implementation exposes programmer-supplied annotations, and or other information about the implementation that facilitates an optimized specialization of

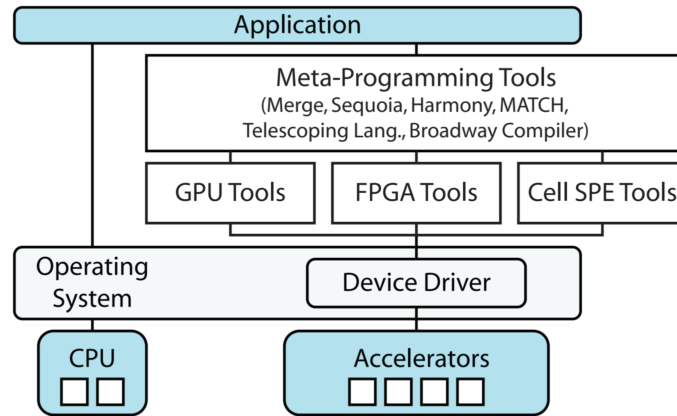


Figure 2.3: Role of library metaprogramming tools

the otherwise generic function call.

Figure 2.3 summarizes the role of metaprogramming tools for heterogeneous systems. Metaprogramming tools are not responsible for directly producing machine code (or its equivalent) for an accelerator(s); instead these tools attempt to smartly select among different specialized implementations, which may use accelerator-specific or application domain-specific languages and compilers, for a particular computation. A smart mapping (sometimes termed library-level optimizations) can include choosing specialized function implementations in place of more general ones (including those targeting specialized accelerators), eliminating unnecessary library calls, moving library calls around, and specializing the function implementation for a particular call site [27].

A flexible mapping requires a level of indirection between the function call and implementation, a feature shared by all the tools shown in Figure 2.3. Mechanisms from existing mainstream languages, such as runtime polymorphism and the template system in C++, could be used to create the necessary indirection; however, as described in the context of the VSIPL++ library (Section 2.2.2), which makes extensive use the C++ templates, existing mainstream languages are insufficient to implement more advanced inter-procedural optimizations [27]. The limitations of existing programming models motivate the development of language features to support more sophisticated metaprogramming.

The typical architecture of a library metaprogramming tool is shown in Figure 2.4. The annotations describe and expose semantic information about function implementations, and may be included within the implementation, as a separate file in the library (shown), or with the application itself. The annotations, along with library headers, and the library and application source code

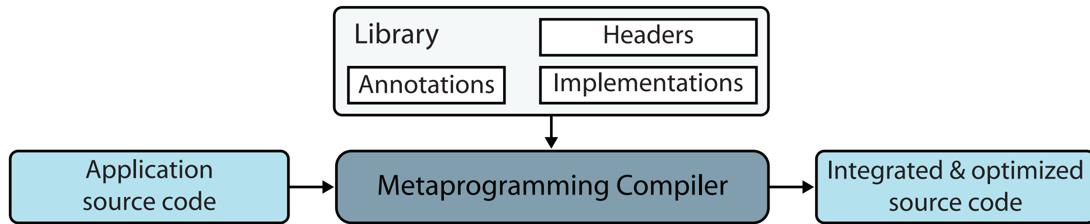


Figure 2.4: Typical architecture of library metaprogramming tools, adapted from [27]

are processed by the metaprogramming compiler, which performs the mapping between application and the library, producing an integrated and optimized application that is then compiled and linked using conventional tools (and any specialized compilers for particular accelerators or application domain-specific languages).

The annotations address the specification gap between traditional compiler primitives and their function equivalents. Traditional compiler primitives are documented in significant detail, and all users of those primitives (tools within the compiler) have full access to those specifications. In contrast, functions are often defined in the application itself, and can have implementations that are opaque to a particular compilation unit. As a result, reasoning about functions is much more challenging for the compiler. The annotations provide an implementation-independent mechanism for the programmer to communicate to the compiler and runtime information needed to safely and smartly optimize across functions.

Within the basic structure shown in Figure 2.4, library metaprogramming tools can differ across a number of dimensions, notably:

Library or Application-based Annotations: Are annotations provided once with the function implementations in the library, or with each application that uses the library.

Descriptive or Prescriptive Annotations: Do the annotations describe the specialized function implementations, or prescribe how the application should be specialized.

Generation Function: Can new specialized function implementations be generated automatically and transparently to the application

Specialized Languages: Does the programming model support implementations using specialized languages, i.e. using languages that differ from the base language.

Dynamic or Static Specialization: Is function specialization performed dynamically at runtime, or statically at compile time.

These differences are summarized for several relevant library metaprogramming tools, along with

Table 2.1: Summary of features of cited library metaprogramming tools. Empty circles indicate partial support for a particular feature.

	Annotations: Library-based	Annotations: Application-based	Annotations: Descriptive	Annotations: Prescriptive	Generate functions	Specialized Languages	Dynamic specialization	Static specialization
Broadway	●			●				●
Telescoping Lang.	●			●	●			●
Sequoia		●		●		○		●
MATCH	○			●	●	●		●
Harmony	○		○			●	●	
Merge	●		●		○	●	●	

the Merge programming model described in this thesis, in Table 2.1. Each programming model is described in more detail in the following sections.

Different design choices across these dimensions can have a significant impact on the effectiveness of a tool for heterogeneous systems. For instance, application-based, prescriptive annotations enable the programmer to precisely map their application to a particular hardware platform. The tradeoff for that level of control is that the application programmer, often a domain expert not a parallel programming guru, is completely responsible for defining a smart mapping for the particular application and input and machine configurations. The programmer needs to be knowledgeable about both the application and the target hardware – a rare combination. Changes to the input configuration, such as much larger or smaller inputs, or the machine configuration, such as new accelerators being installed, may require the programmer to change the annotations, negatively impacting portability or maintainability.

Static specialization can similarly negatively impact portability and maintainability when the input or machine configuration changes. Static specialization forces runtime specialization for input configuration or accelerator availability into the function implementation itself, typically in the form of `if-else` statements, reintroducing many of the limitations of library-based programming (discussed in Section 2.2.2) that metaprogramming was supposed to address. The tradeoff for runtime specialization is the potential additional overhead for dynamic function specialization.

The additional challenges and requirements for library metaprogramming tools for heterogeneous systems, such as the limitations in static specialization, are described in more detail in the following sections in the context of particular tools. Note though that not all of these tools were designed for heterogeneous systems, or for the same usage scenario, non-expert programmers using hardware resources shared with other applications or users, as Merge. The following discussion is not intended as a criticism of the design choices made for these tools, instead the goal is to orient the reader to different points in the design space and motivate the choices made for the Merge programming model. Further note that following descriptions focus on the features of each tool that are relevant for heterogeneous systems, and do not attempt to provide a complete presentation of each system's various capabilities and features.

Broadway Compiler

The Broadway Compiler [27, 26] provides a static function metaprogramming system for C applications and libraries using prescriptive annotations targeting GPPs. The library developer includes a separate annotation file alongside the library header and source code that provides a set of optimizing transformations for the library functions. The transformations are implemented using a sophisticated metaprogramming language that can test different properties of the application code, including dataflow dependencies, control-flow context, and abstract interpretation results. At compile time the tests are evaluated and the specified action, which might move, remove or substitute matching code, is applied as appropriate.

An example transformation, adapted from [27], for a function that splits a matrix into two smaller submatrices (upper and lower) is shown in Figure 2.5. The two transformations (lines 6 and 7) specialize the implementation of the splitting function based on the memory distribution of the input matrix. Both transformations prescribe how to specialize the splitting function, as opposed to describing the properties of any particular specialized implementation. Directly expressing the transformations gives the library developer complete control over the specialization process, but at expense of added difficulty in introducing new implementations. Integrating a new specialization for a function requires that the library developer modify the existing transformations for that function. To do so, the developer must understand the relative strengths and weaknesses of both the new implementation and the existing versions, a potentially non-trivial undertaking.

Transformations are statically evaluated. The Broadway compiler includes compile-time dataflow analysis to characterize the application prior to the transformations. The static specialization avoids introducing conditionals, *e.g.* `if-else` statements, and other code that would introduce additional

```
  pattern stmt{
    PLA_Obj_horz_split( ${obj:A}, ${expr:size}, ${obj:upper_ptr}, ${obj:lower_ptr})
    }stmt
  {
5   on_entry { A --> __view_1, DATA of __view_1 --> __Data }
    when (Distribution of __view_1 == Empty) remove;
    when (Distribution of __view_1 == Local)
      replace stmt{
        PLA_obj_view_all($A, $upper_ptr);
10    }stmt
  }
```

Figure 2.5: Example metaprogramming annotation for the Broadway compiler defining the specialization for function based on the memory distribution of the inputs.

runtime overhead. For heterogeneous systems, particularly those using GPUs and other accelerators, which might be shared among multiple clients in the system, some type of runtime specialization, responsive to the dynamic availability of the accelerator, is required. Runtime specialization is also required for parameters, such as input matrix size, that cannot be known statically but might determine which specialized implementations might be invoked. For tools with static specialization, runtime transformations are pushed into the application or library code, with all the attendant disadvantages described in the context of library-based programming.

The combination of sophisticated dataflow analysis and a pattern matching metaprogramming language enables a wide range of potential optimizations. However, as the code sample in Figure 2.5 shows, the annotations are complex. Generating the appropriate annotations can be a non-trivial undertaking with a level of effort potentially on par with implementing the function itself.

Telescoping Languages

Telescoping languages [36] provide aggressive library synthesis for high-level scripting languages, particularly for Matlab and the R statistical programming environment, targeting GPPs. Like the Broadway compiler, Telescoping languages include sophisticated compile-time dataflow analysis of the high-level scripts. The characterization provided by this analysis is used to control the compile-time specialization of the function library. The primary difference between the two is Telescoping languages actively generate specialized function implementations for known application domains.

Built into the Telescoping compiler is a sophisticated understanding of the application domain, *e.g.* Matlab's matrix operations. Based on the analysis of a Matlab library function, the compiler identifies valid size configurations (*e.g.* scalar, matrix) of the inputs. For each configuration, the

compiler generates specialized implementations using existing C or FORTRAN linear algebra libraries, such as BLAS, for both those particular input sizes and other matrix properties, such as symmetry, that might enable more efficient algorithms. The generation process is performed once for each library function. When that function is used in a script, similar dataflow analysis is performed on the entire script to select one of the specialized implementations.

Numerous other tools provide synthesis of specialized libraries. The ATLAS [67] and FFTW [22] libraries are implemented with parameterized algorithms that can be automatically tuned for particular platforms using a heuristic-guided search through a domain-specific parameter space (matrix operations and FFT, respectively). The Intel C++ compiler [33] automatically generates and transparently selects among multiple variants of compiled code, specific to certain architectural features like SSE. Unlike many of the metaprogramming tools described this section, these libraries and the Intel compiler do not expose the metaprogramming process to the programmer, preventing them from integrating additional implementations for input or machine configurations not considered by the original developers. As such, libraries are often used as building blocks for other higher-level metaprogramming tools, such as Telescoping languages, which attempt to further increase performance by optimizing across specialized function calls.

Although Telescoping languages are more general than the specialized libraries cited above, they are still domain-specific, and provide only limited opportunities for the programmer to extend the metaprogramming process. The library developer can provide hints about functions to the metaprogramming compiler, but that appears to be the extent of the user's influence. Keeping pace with the rapid evolution of heterogeneous systems requires the programmers to have more influence over the metaprogramming process, and in particular be able to more easily integrate new specialized implementations for cutting-edge hardware.

Sequoia

In contrast to the Broadway compiler and Telescoping languages, the Sequoia [20] programming model exposes the metaprogramming process entirely to the application programmer. The application developer provides a separate specification file with the application that explicitly defines how generic function calls in Sequoia applications should be specialized.

The specification file also provides values for "tunable" parameters that are unbound in the Sequoia source code. Specialization is static, manual and completely prescribed by the specification file. An example implementation for matrix multiplication using in Sequoia is shown in Figure 2.6, along with the programmer-supplied specification file that specializes the implementation for the

```
void task matmul::inner(in float A[M][P],
                        in float B[P][N],
                        inout float C[M][N])
{
 // Partition matrices into 2D blocks
 tunable int U, X, V;
 blkset Ablks = rchop(A,U,X);
 blkset Bblks = rchop(B,X,V);
 blkset Cblks = rchop(C,U,V);

 // Compute C in parallel
 mappar (int i=0 to M/U, int j=0 to N/V)
  mapreduce (int k=0 to P/X)
   matmul(Ablks[i][k],Bblks[k][j],Cblks[i][j]);
}

void task matmul::leaf(in float A[M][P],
                       in float B[P][N],
                       inout float C[M][N]) {
  for (int i=0; i<M; i++)
    for (int j=0; j<N; j++)
      for (int k=0; k<P; k++)
        C[i][j] += A[i][k] * B[k][j]
}
```

(a) Dense matrix multiplication in Sequoia

```
instance {
  name     = matmul_mainmem_inst
  task     = matmul::inner
  runs_at  = main_memory
  calls    = matmul_LS_inst
  tunable U=128, X=64, V=128
}
instance {
  name = matmul_LS_inst
  variant = matmul::leaf
  runs_at = LS_level
}
```

(b) Specification for `matmul` on the Cell processor

Figure 2.6: Example implementation of dense matrix multiplication in Sequoia (a), along with specification file specializing the implementation for the Cell processor (b). Adapted from [20].

Cell processor. The tunable parameters in the specification shown in Figure 2.6b line 6 partition the matrices into sub-blocks that will fit in the local stores of the Cell processor's 8 synergistic processing elements (SPEs); while the `variant` annotation indicates the particular function implementation that should be invoked by the `matmul` call in Figure 2.6a line 14.

Application programmers using Sequoia have complete control over the specialization process. However, effectively using that control when developing or maintaining an application requires that the programmer know about and select from the set of potential specialized implementations. Existing applications cannot take advantage of new specialized implementations until the programmer has updated their specification file to integrate the new function variant. For example, to integrate a new matrix multiply implementation for the Cell SPEs, the programmer needs to both update the call specification (`calls` in the above specification) and ensure that all tunable parameters are compatible with the new variant.

Sequoia is designed for platforms, such as clusters and the Cell processor, with hierarchal symmetrical distributed memory systems. For these systems, efficient movement of data up and down the memory hierarchy is critical to achieving good application performance. Each platform has a

slightly different memory hierarchy, however. To abstract away the heterogeneity and complexity of this data movement, the actual data transfers are implicit in Sequoia programs. Sequoia's support for heterogeneous systems is limited however to those with similar hierarchical memory systems. And since the compiler automatically inserts the code needed to effect the data transfers, the Sequoia compiler and runtime must include significant platform-specific components.

MATCH

The MATCH compiler [46] maps Matlab scripts to optimized libraries developed for a heterogeneous computing environment consisting of embedded GPP processors, DSP processors and FPGAs. For functions provided in Matlab, such as matrix multiplication or FFT, the compiler writers supply implementations for one or more of the specialized accelerators. Functions written in Matlab by the application programmer are automatically compiled for the different processors. At compile time, the application is statically scheduled on the heterogeneous computing environment using a linear programming-based scheduling algorithm and estimates of the execution time of each function implementation. The programmer can affect the scheduling process through prescriptive annotations included in the Matlab application.

The MATCH compiler appears to be closely tied to a particular heterogeneous computing environment and a particular set of specialized function implementations. The published papers do not describe any language or tool support for adding new function specializations. It is not clear how the applications could or would be ported to a different platform and different function implementations. And if you could integrate new implementations it appears MATCH would present many of the same issues as other metaprogramming tools with static, prescriptive, application-based annotations.

Harmony

The Harmony [16] programming model and runtime provides a processor-like out-of-order (OoO) execution model for heterogeneous systems. Functions, termed compute kernels, are scheduled like instructions in a modern processor. The application thread dispatches kernels in a non-blocking fashion using the Harmony API. The Harmony runtime queues the function calls, along with dependence information, in a dispatch window (analogous to the instruction window in a processor). Kernels are dispatched dynamically from this window to the different computing resources in the

systems based on the availability and relative performance of the available function implementations. Much like modern processors, Harmony supports prediction and speculative execution to expose parallelism in the presence of control flow.

Unlike the other metaprogramming tools described in this section, Harmony does not provide any annotation support for the programmer to control function call specialization. To ensure correctness, Harmony requires that specialized function implementations targeting different hardware resources be completely interchangeable across all possible input configurations. For many kernels it might not be possible to build efficient or correct variants across the entire possible input space. For discrete GPUs, for example, the kernel must perform enough computation to amortize the overhead of data transfers between the CPU and GPU. As a result most kernels will have a lower bound at which they can be efficiently offloaded to the GPU. Harmony uses performance guided scheduling to attempt to detect and avoid invoking inefficient function implementations. The programmer must still, however, invest the effort to ensure that every function implementation returns correct results across the entire input space, even for those input configurations for which an implementation should not, and likely will not, be invoked.

Many streaming implementations do this odd issue. — After reading this, I still don't have much of a sense of what harmony is, or how it works

So what does this all mean? I generally don't like chapters to end with details — I think some bridge to what comes next.

Chapter 3

Merge Programming Model

The Merge programming model seeks to make complex heterogeneous systems accessible to informaticians and other domain experts. The combination of user's often limited programming experience, rapidly evolving applications and hardware, and the need to support platforms where hardware resources are shared among multiple applications or users, introduces additional challenges and requirements for library metaprogramming tools. As enumerated in Chapter 2, library metaprogramming tools for shared heterogeneous systems should: (1) support multiple different accelerator-specific or application-domain specific programming models; (2) readily integrate new specialized function implementations, targeting new or different hardware, without changing the application; and (3) enable dynamic specialization of function calls.

The Merge programming model is specifically designed to address these challenges and requirements. Using the terminology defined in Chapter 2, Merge is a dynamic library metaprogramming system using descriptive, library-based annotations.

This chapter describes the Merge programming model in detail. Section 3.1 describes the three key characteristics and capabilities of Merge, encapsulation, interchangeability, and bundling, and how they address the challenges described above. Sections 3.2 and 3.3 show how encapsulation and bundling can be used to manage resources in complex heterogeneous systems, enable aggressive compiler optimizations, and support different scheduling techniques.

3.1 Core Characteristics of Merge

Encapsulation, interchangeability, and bundling are the core capabilities and characteristics of Merge and are described in more detail in the following sections.

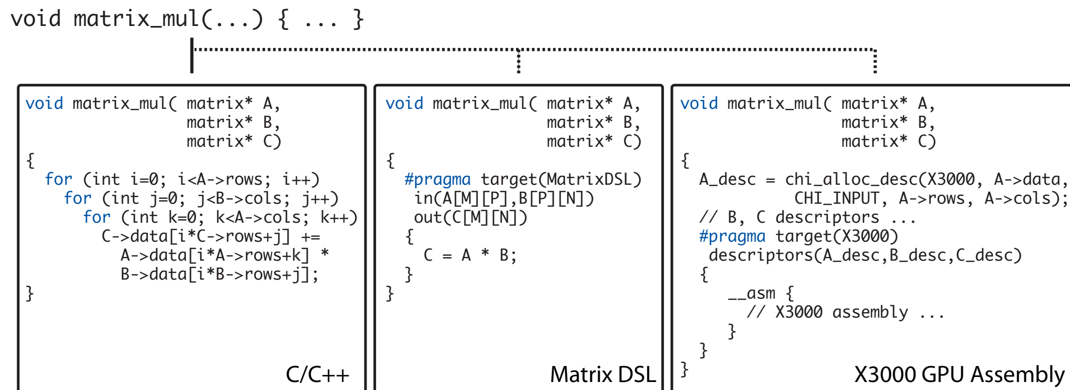


Figure 3.1: Encapsulation of inline accelerator-specific assembly or domain-specific languages

3.1.1 Encapsulating Specialized Code

Merge encapsulates specialized languages targeting specialized accelerators in traditional C/C++ functions to provide a uniform, legacy compatible interface, independent of target architecture. Programmers can build applications much as they do now, by targeting existing APIs, or creating new APIs and implementing those interfaces with C/C++ code targeting a general-purpose processor (GPP). When possible and appropriate, however, the programmer can extend his or her those APIs with specialized implementations targeting GPUs, FPGAs and other accelerators. Figure 3.1 shows an example in which `matrix_mul` is implemented using a combination of standard C, a domain-specific language (DSL) and GPU-specific assembly. The three versions present the same interface, but each provides a different choice of implementation to the programmer, compiler and runtime.

Merge implements encapsulation using EXOCHI [64] and other similar tools. EXOCHI enables the creation of a function API for specialized accelerators, even those without sophisticated compiler support, by inlining accelerator-specific assembly or domain-specific language into conventional C/C++ functions. A sketch of the operation of EXOCHI is shown in Figure 3.2 and the C for Heterogeneous Integration (CHI) compilation flow in Figure 3.3. The resulting C/C++ code is largely-indistinguishable from existing ISA intrinsics (like those used for SSE instructions), enabling the programmer to create new intrinsics, termed *function-intrinsics*, of any complexity for any architecture that supports the lightweight Exoskeleton Sequencer (EXO) interface[1].

EXO supports a shared virtual memory multi-threaded programming model for heterogeneous

[1]Other, vendor-specific tools like NVidia's CUDA, have similar capabilities; to focus the discussion, however, I will exclusively discuss on EXOCHI, on which the Merge prototype described in Chapter 4, is built. The techniques presented are equally applicable to other tools, such as CUDA.

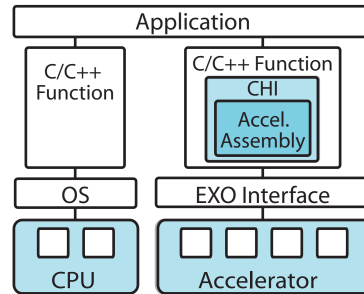


Figure 3.2: Sketch of EXOCHI framework

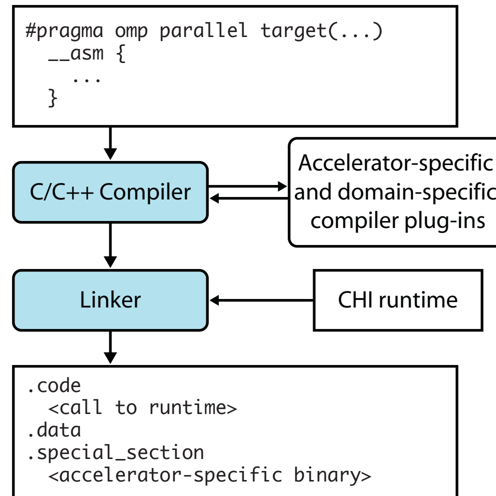


Figure 3.3: CHI compilation flow (adapted from [64]).

systems. The heterogeneous cores are exposed to the programmer as sequencers [28], on which user-level threads, encoded in the accelerator-specific ISA, execute. EXO serves as a type of proxy layer, supporting interaction between the specialized accelerator and the CPU by initiating or responding to inter-sequencer user-level interrupts. In particular, EXO supports proxying actions that require OS support, such as page table updates or exceptions, on the CPU on behalf of the specialized accelerator.

Using CHI, the programmer implements function-intrinsics for specialized accelerators by inlining accelerator-specific assembly or code written in a domain-specific language into conventional C/C++ code. CHI example code is shown in Figure 3.4. CHI enables the programmer to define a

```
A_desc = chi_alloc_desc(X3000, A, CHI_INPUT, n, 1);
B_desc = chi_alloc_desc(X3000, B, CHI_INPUT, n, 1);
C_desc = chi_alloc_desc(X3000, B, CHI_OUTPUT, n, 1);
#pragma omp parallel target(X3000) shared(A,B,C)
  descriptor(A_desc,B_desc,C_desc) private(i)
{
  for (i=0; i<n; i+=8)
    __asm
    {
      ld.8.dw  [vr0..vr7]   = (A, i, 0)
      ld.8.dw  [vr8..vr15]  = (B, i, 0)
      add.8.dw [vr16..vr23] = [vr0..vr7], [vr8..vr15]
      st.8.dw  (C, i, 0)    = [vr16..vr23]
    }
}
```

Figure 3.4: CHI code example for vector add with Intel X3000 integrated GPU pseudo-code. Adapted from [64].

uniform C/C++ function interface that is independent of the actual implementation and fully compatible with existing CPU-based software infrastructure.

When the programmer uses a non-C language, such as the GPU-specific assembly shown in Figure 3.4, the appropriate accelerator-specific or domain-specific compiler, determined from the `target` clause in the `#pragma`, is invoked. The resulting machine code, produced by the external compiler, is embedded in the special section of the CPU executable, along with the necessary runtime calls to dispatch the specialized machine code. A sketch of the CHI compilation flow is shown in Figure 3.3. The final product is a "fat" binary with multiple code sections for different heterogeneous processors.

From the programming language perspective, the primary role of EXOCHI is to provide the necessary interface for a semantic transition between the C-like execution model and a more specialized accelerator-specific or domain-specific execution model. In the case of the example in Figure 3.4, using the Intel X3000 GPU, the `#pragma` statement is expanded to a series of calls into the X3000 driver to setup data descriptors, transfer data to and from the GPU's memory space and load the GPU's command queue. In other cases, the transition might be more logical than physical. In the MatrixDSL example in Figure 3.1, the `#pragma`'s primarily role is to redefine the in/out arguments as matrices understood by the MatrixDSL (although such a transition might also include performing both compile-time and runtime checks on the arguments). The syntactic and semantic flexibility of `#pragmas` enables Merge to use a common abstraction for the transition between many different execution models.

Using EXOCHI (or a similar tool) decouples the support for specialized accelerators or programming models from the support for metaprogramming. Although the Merge compiler can "peek" into encapsulated code for optimization purposes, it does not need to do so for basic operation. As result, new encapsulated toolchains, provided by the accelerator vendor or others, can be integrated into Merge with only minimal changes to the Merge compiler and runtime itself. To enable optimization across functions, though, even across opaque encapsulated implementations, Merge places restrictions on function implementations and provides implementation-independent annotation mechanisms, both described in the following sections.

3.1.2 Interchangeable Concurrent Functions

The simple definition of the C/C++ function call ABI provides a reasonable starting point for function-intrinsics, but must be enhanced to provide guarantees needed for correct execution in a heterogeneous multicore environment. Additional restrictions are required to ensure that different implementations of the same function, like those shown in Figure 3.1, can be invoked interchangeably, independently and potentially concurrently. These restrictions fall into two categories: (1) those needed for correct concurrency; and (2) those needed to ensure implementations targeting specialized accelerators can be invoked interchangeably with their GPP-only counterparts.

Concurrency-driven Restrictions

All function-intrinsics are required to: (1) be independent and potentially concurrent; (2) only access data passed as arguments; and (3) execute atomically with regards to each other. Function-intrinsics can directly communicate with other function-intrinsics only through function call and return operations. These restrictions are similar to the task-based execution model used in Sequoia [20]. Unlike Sequoia, however, sibling function-intrinsics, those functions invoked by the same parent, may communicate indirectly through their arguments and a function-intrinsic can have dependent function-intrinsics other than its direct parent. In this way dataflow dependencies between function-intrinsics can be enforced without spawning.

Not sure what this means.

As an example of indirect communication, consider the computation of the similarity matrix used by the Smith-Waterman sequence alignment algorithm [59]. Smith-Waterman is a dynamic programming algorithm in which each element of the similarity matrix, $S_{i,j}$ is dependent on $S_{i-1,j-1}$, $S_{i,0:j-1}$ and $S_{0:i-1,j}$. Independent entries in the similarity matrix can be computed in parallel; however, enforcing the dataflow dependencies using just function call and return is tedious

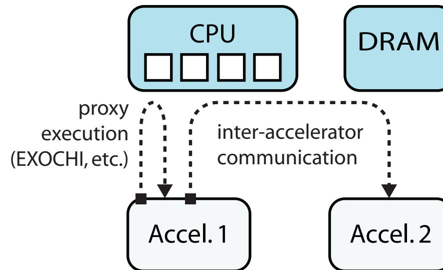


Figure 3.5: Sketch of communication between accelerator(s) and CPU(s), which acts as a hub

and complicated. Instead the necessary synchronization can be built into the matrix data structures in the form of futures. The function-intrinsic computing the entire similarity matrix can partition the matrix and invoke child functions for each patch in parallel. When attempting to read values that have not yet been produced by its siblings, a function-intrinsic would block, and yield control of the processing element until the desired data is available.

Sequoia's limitations on sibling-to-sibling interaction are necessary to statically analyze the communication between different invocations of a function-intrinsic executing on a platform with a distributed memory system. However, on a shared-memory platform, for instance, Sequoia's execution model can be unnecessarily restrictive. To be more inclusive, the Merge programming model uses a more flexible execution model that allows concurrently executing function-intrinsics to indirectly communicate through their caller's context.

Interchangeability-driven Restrictions

Legacy applications and software infrastructure assume computation occurs on the CPU(s), or more specifically in the CPU's shared virtual memory space. Thus for specialized function-intrinsics to be transparently interchangeable with their legacy CPU-only counterparts, the computation must appear to the caller to start and complete on the CPU and in its memory space, regardless of where the computation actually occurs. Accordingly, Merge defines the CPU and its memory space to be the hub of the system, shown in Figure 3.5. This organization reflects the typical construction of heterogeneous systems, in which the general-purpose CPU coordinates activities throughout the system.

Function-intrinsics targeting processors in other memory spaces, such as GPUs, must ensure that the necessary data is copied between the CPU and the accelerator. This can be performed explicitly by programmer, or using a DSL, like Sequoia, with support for distributed memory systems, or

using EXO to provide the appearance of a single shared memory space (or some other way). For example, in the sample code in Figure 3.4, lines 1-3 define data descriptors that are used by the EXOCHI runtime to transfer data between the CPU and GPU memory spaces.

All function-intrinsics callable from the CPU's context must observe the above restrictions; those functions that are only callable from an accelerator context do not. In the latter cases no data copying or other interface code is needed. Programmers might build such a function to reuse the same specialized code in different scenarios. Accelerator-specific reuse is quite common in programming environments, such as NVidia's CUDA, that provides this capability. In CUDA, for example, the programmer can mark some functions targeting the GPU as `global`, indicating that they are callable from the CPU context, and others as `device`, indicating that they can be only called from other functions that will execute on the GPU.

3.1.3 Function Bundling

Since any one implementation of a function is unlikely to be the most efficient, or even usable, for all input and machine configurations, multiple implementations should be able to coexist. These might include specialized function-intrinsics, like those shown in Figure 3.1, targeting different hardware resources, using different programming languages or parameterized for different input scenarios. Collecting together multiple function-intrinsics optimized for different inputs and hardware provides a rich set of potential implementations from which the compiler and runtime can choose to improve performance and efficiency.

Selection is typically implemented using a combination of static and dynamic techniques. Narrowing the choice of the function-intrinsics to those targeting installed hardware is typically performed through selective compilation (`#ifdef`) and static or dynamic linking. The C standard library, for instance, is specialized through system-specific linking. Choosing a function-intrinsic that is efficient for particular inputs and smartly distributes work across a heterogeneous systems is traditionally performed using dynamic (linking, `if-else`) techniques. When there were only a few target architectures available, the above approaches were less problematic, but as the diversity of heterogeneous systems grows, and systems with multiple accelerators become commonplace, the combinatorics will make such approaches impractical. The problem is particularly acute if the programmer must manually choose among the available implementations.

Merge replaces the current ad hoc mix of static and dynamic selection with a unified approach built around predicate dispatch [18]. Predicate dispatch subsumes single and multiple dispatch, conditioning function invocation on a boolean predicate function that can include the argument types,

But its implementation must at some level go through a bunch of decision steps. I am not sure why this will be much faster dispatch. I can convert if-then-else to dispatch table too, right?

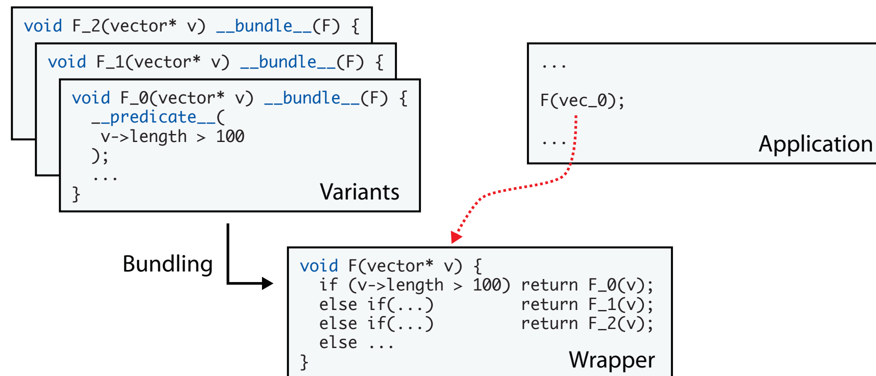


Figure 3.6: Sketch of function bundling and example meta-wrapper usage

values and system configuration. A programmer supplies a set of annotations with each function implementation. These annotations provide a common mechanism for describing the invariants for a given function, independent of the programming model used to implement the particular function intrinsic. To improve clarity the following paragraphs provide a high-level description of the Merge dispatch system; Chapter 4 describes the prototype implementation of function bundling in detail.

There are three classes of annotations: (1) *input restriction*, which are boolean restrictions on the input arguments (*e.g.*, data set size < 10000); (2) *configuration restrictions*, which specify the necessary compute resources (*e.g.*, 10 kB of scratchpad memory); and (3) *traits*, which describe properties that are useful to consumers of the function (*e.g.*, associativity). At compile time, when function-intrinsics implementing the same computation are bundled together, the annotations are analyzed and translated into meta-wrapper functions that implement the generic function interface, while providing a level of indirection between the caller and the different function variants. Bundling and meta-wrapper usage is sketched in Figure 3.6. The meta-wrappers are not limited to simple dispatch wrappers, like the one shown on Figure 3.6; "lookup" and other introspective meta-wrappers, which enable applications to query the availability of specialized function variants, can and are generated from the function bundles.

The mapping of architecture-agnostic applications to architecture-specific function intrinsics is determined by some objective function. When traditional techniques, such as selective compilation and manual dispatch code, are used, the implementation of that objective function is embedded directly in the application itself. Applications built in this way are challenging to maintain, and are difficult to extend to exploit new or different heterogeneous systems. Prescriptive library metaprogramming systems can have similar limitations. Specifying function specialization prescriptively,

and outside the function implementations (*e.g.* in the application, or in a separate file in the library) requires that the programmer modify these specialization specifications when incorporating new implementations. In contrast adding a new function-intrinsic to a function bundle does not require the programmer to manually modify existing code, only to regenerate the meta-wrappers.

Function bundling separates the application from the implementation of the mapping function, easing maintenance and extension. The bundler uses the descriptive annotations, along with profiling and other techniques, to automatically construct the objective function and thus the mapping between the application and the function-intrinsics. The tradeoff is that the compiler and runtime must infer a good objective function for a particular application and machine configuration. The performance results in Chapter 4 show that programmers can effectively exploit heterogeneous systems, even using very simple inferred objective functions. For those programmers and programs that require more control, however, alternative specialization approaches could be used. One example, described in the following section on usage models, shows how the Harmony [16] runtime could use Merge's function bundles. In this case, Harmony would use meta-wrappers that return pointers to all applicable implementations, instead of just one. Alternately a completely static programmer-defined specialization mechanism, similar to Sequoia's specialization file, could be provided. However, by making automatic specialization the default, non-expert programmers are not obligated to immerse themselves in the details of the particular specialized function implementations that might be available.

Similar reasoning applies to the choice of dynamic specialization. Although static specialization eliminates a potential source of overhead, alternate mechanisms must be provided for those specialization decisions, such as accelerator availability, that cannot be made until runtime. Dynamic specialization enables the use of a common mechanism, in this case, predicate dispatch. Existing compiler optimizations, such as constant propagation and inlining, can be used to simplify, and potentially eliminate, the dispatch wrappers when parameters are known at compile time. When the dispatch wrapper overhead cannot be ignored, static specialization, like that described above, could be used to eliminate wrappers. In general, significant dispatch overhead indicates that application is being specialized at too fine a granularity. In this situation there are alternate techniques, described in the following sections on library construction, that do not require switching to static specialization.

3.2 Usage Models

In the simplest usage model of Merge, the application programmer does not need to be aware that any particular function call might invoke a specialized function-intrinsic targeting a non-GPP computing resource. They simply call a function, *e.g.* `matrix_mult`, as they would normally. The restrictions specified in Section 3.1.2 ensure that all implementations in a function bundle are interchangeable (subject to the invariants specified by their annotations). This usage model could be described as the "direct" model, in which the use of Merge is largely transparent to the programmer. There is also an "indirect" model, where the programmer either explicitly or implicitly uses the introspection and indirection offered by function bundles to implement advanced functionality on top of the core Merge capabilities.

The Harmony runtime scheduler, described in Chapter 2, could, for instance, be redesigned as an indirect user of the Merge programming model. Presently, at scheduling time the Harmony runtime computes the intersection between the processors installed in the system and the implementations available for a kernel to determine the set of processors that could potentially execute the kernel. Harmony then schedules the kernel onto one of the processors from this set using a first-to-finish metric. Merge's predicate-driven function bundles provide a much more comprehensive approach to determining the available specializations. Instead of a single predicate, processor architecture, the intersection operation can include an arbitrary set of conditions on the inputs to the kernel and the machine configuration. In this usage model, the Harmony runtime would explicitly query the function bundles for all applicable implementations, and then choose among them based on its own scheduling algorithm.

The prototype implementation of Merge, described in Chapter 4, uses function bundle introspection to implement an encapsulated map-reduce-based parallel programming language. The Merge runtime speculatively queries for coarse grain function implementations that would compute multiple iterations of the map and reduce function in one invocation. Such coarse grain variants minimize runtime overhead, while offering a more specific translation of the generic parallelism expressed in the map-reduce representation to a particular parallel implementation for a given hardware resource. If a coarse grain variant is not available, the runtime defaults to directly executing the map-reduce statement. Using the introspection and indirection provided by function bundles, the Merge runtime can readily implement a more sophisticated scheduler for the map-reduce language.

A combination of the direct and indirect usage models enables application,compiler, and runtime developers to choose just how much control they will have over the specialization process.

The simple dispatch wrappers provide a low-complexity, low-effort path for non-expert developers wishing to use specialized implementations. Expert developers that require more control, or wish to implement more sophisticated domain-specific scheduling techniques, can use the indirect meta-wrappers to explicitly choose among the available implementations. With this flexibility, the function bundles will not inhibit any programmer from building an application that satisfies their requirements. And for many, language support for multiple implementations and dispatch meta-wrappers will reduce the difficulty of developing applications and tools for complex heterogeneous systems.

3.3 Library Construction

An application using Merge is constructed as a hierarchical set of functions that break the computation into successively smaller operations. The programmer determines at which step in the hierarchy, *i.e.* at what granularity, the computation might best be offloaded to particular accelerator (again, subject to the constraints in Section 3.1.2). For example, to profit from offloading a computation to a discrete GPU, the computation must have enough arithmetic operations to amortize the latency of the data transfer to and from the GPU. Identifying the right granularity at which to offload computation to a specialized accelerator, or in other terms, identifying the granularity at which a function-intrinsic is efficiently interchangeable with other implementations, is a key challenge for heterogeneous systems.

Flexible encapsulation and function bundling provide the necessary infrastructure to enable the programmer to create function-intrinsics that are efficiently and effectively interchangeable. The following sections describe two common scenarios, 1) smartly structuring computation for particular computing resources and 2) smartly assembling hardware resources for a particular computation, that arise when using heterogeneous systems, and how the library programmer would use Merge in these situations.

3.3.1 Structuring Computations for Particular Heterogeneous Hardware

Consider the simple example pseudo code shown in Figure 3.7, in which two functions are called in sequence. In the simplest usage, the `F` and `G` function calls could be independently mapped to different hardware resources, with data copied between the CPU and accelerator memory space as needed. For some inputs, the overhead of the data copying will be adequately amortized, and this approach will be satisfactory. Annotations, supplied by the programmer or generated through some

```
matrix H(matrix A, matrix B, matrix C) {
  matrix T1 = F(A, B);
  matrix T2 = G(T1, C);
  return T2;
}
```

Figure 3.7: Example function that might be superseded by an alternate implementation that eliminates CPU–accelerator communication while maintaining interchangeability.

type of performance profiling, can be used to limit the invocation of a particular to just those inputs for which it will be effective.

In many instances performance can be improved by eliminating the intermediate data transfers (of `T1`). However, independent implementations for `F` or `G` cannot forego copying `T1` and remain interchangeable. Doing so would be making an implicit assumption about which memory space contained the valid copy of `T1`. That assumption can be made explicit, however, by creating a new implementation for `H` that includes the computations in both `F` and `G` and is interchangeable with the other implementations of `H`. In effect, the programmer is inter-procedurally optimizing across `F` and `G` to create a new version of `H` that can be bundled alongside the version in Figure 3.7 and any other implementations.

When there is little or no sophisticated compiler support, there is no other option than for the programmer to manually build up optimized implementations. Many of the function-intrinsics developed for the Intel X3000 GPU used in the evaluation system described in Chapter 4 were implemented in this way. Functions were fused together until the function-intrinsic performed enough computation to amortize the data transfer latency. As compiler support improves some of these optimizations will be automated. Merge's encapsulation and bundling capabilities facilitate these inter-procedural optimizations. Encapsulated languages provide the input for the optimization, with the product, and its associated dispatch annotations, integrated into the function bundles as an alternated implementation. Such automatic optimizations are a topic of future research.

3.3.2 Assembling Hardware Resources for Particular Computations

Extensive resource virtualization in commodity GPPs has allowed programmers to largely ignore resource management. However, the hardware required for virtualization, such as translation lookaside buffers (TLBs), is expensive and rarely implemented in accelerators, such as GPUs. For example, programmers using NVIDIA's CUDA programming model must explicitly manage the GPU's

local scratchpad memory. For the same efficiency reasons, embedded systems often do not virtualize hardware resources; programmers must explicitly allocate resources, such as memory and bandwidth, in modern heterogeneous systems-on-a-chip.

Merge's encapsulation and bundling capabilities enable the programmer to manage complex software-controlled resources. The proxy layer (such as EXOCHI), described in Section 3.1.1, allows resources that an OS normally considers to be independent to be grouped into a single entity, in which most of the resources are explicitly managed by the programmer [65]. The `configuration` annotations enable programmers to tell the compiler and runtime what resources are required for each function-intrinsics. And specialized function-intrinsics targeting these grouped resource sets can be encapsulated like any others.

The Merge prototype, described in Chapter 4, uses the function bundles to provide a very simple form of resource management for the Intel X3000 integrated GPU, which has only limited virtualization support. The annotations specify that the GPU must be available for a GPU-targeted implementation to be invoked. If, at dispatch-time, the GPU is in currently in use, the dispatch system will default to an alternate implementation in the bundle that does not use the GPU. The dispatch system ensures that only one computation is dispatched to the GPU at a time (something that the driver did not enforce) while the function bundles provide transparent "fallback" if resources, like the GPU, are not available. good, but might not be the best scheduling, right?

The simple resource management implemented in the Merge prototype will probably suffice for many users. However, for those programmers that need more control, Merge provides a foundation for building and integrating into applications function-intrinsics that target more specific sets of resources. By collecting otherwise independent resources together to create units that are allocated and scheduled as a single resource, programmers can create interchangeable implementations that preserve the conventional CPU-centric architecture, as shown in Figure 3.5, and leverage existing software infrastructure such as the OS, while exploiting inter-accelerator interconnect and other difficult to virtualize resources. The more sophisticated resource management described in the following paragraphs is a topic of ongoing research and is not implemented in the prototype. It is included here, however, to further motivate the use of encapsulation and bundling by showing how they can be combined to address resource management challenges (a question that often arises in discussions of the Merge programming model).

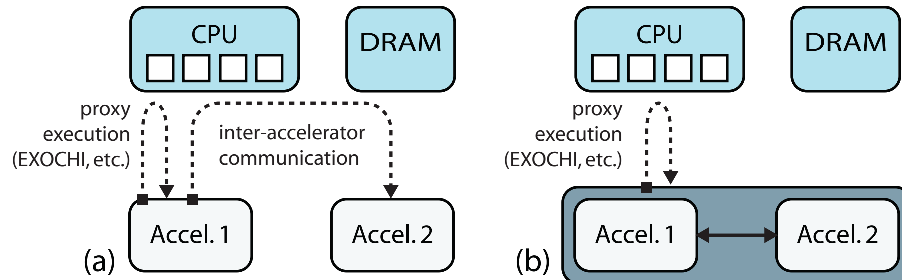


Figure 3.8: Combining accelerators (a) to create new resource sets or achieve performance guarantees, or exploit dedicated communication resources (b)

GPP-like virtualization is ineffective for heterogeneous systems. In the current model, the programmer can only control a virtualized time-slice on a single core, which is insufficient for managing small software-controlled memories or bandwidth to a shared resource, such as crypto accelerator. To efficiently exploit diverse hardware resources, programmers need to be able to assemble more complex hardware resources. Consider the system shown in Figure 3.8b in which otherwise independent accelerators in Figure 3.8a are enhanced with a dedicated, non-virtualized, communication link. Potentially, significant performance improvements could be achieved by using that link for inter-accelerator communication, instead of transferring data through the CPU. Correct execution requires appropriate function-intrinsics are invoked on both accelerators, at the same time; an invariant that cannot be guaranteed if the accelerators are treated independently. Instead the programmer must treat both accelerators as a single unit, forming a new, more complex, *resource set*, on which a single specialized function-intrinsic is invoked.

Flexible resource sets allow programmers to assembly multiple, otherwise independent resources in a single unit when needed. Each resource set can be considered a unique hardware resource, and perhaps even a different class of processor (that might favor a different programming model). For example, tiled architectures, such as MIT's RAW [62], might be treated as 64 independent tiles and programmed using existing threading frameworks (*e.g.* POSIX threads) or might be treated as a single coordinated systolic array and programmed using a streaming language [25]. And between these extremes there are usage models that blend the high-level streaming language with custom-implemented kernels that use low-level threading primitives. Encapsulation and function bundling provide the necessary infrastructure to support flexible resource sets.

Chapter 4

Merge Compiler and Runtime

A prototype of the Merge compiler and runtime has been implemented, targeting desktop scale heterogeneous systems. The Merge prototype consists of three components: (1) an encapsulated parallel domain-specific languages (DSL) based on the map-reduce pattern; (2) a practical predicate-dispatch system implementing the function bundling described in Section 3.1.3; and (3) a compiler and runtime which implement the map-reduce pattern by dynamically selecting the best available function-intrinsics. This chapter describes the design of Merge prototype, and presents a experimental results for the Merge implementation on both heterogeneous and homogeneous platforms.

4.1 Merge Prototype Design

Merge uses EXOCHI [64] to implement function-intrinsics on specialized accelerators, particularly GPUs. The combination of the Merge predicate-dispatch system and EXOCHI enables programmers to transparently substitute function-intrinsics targeting specialized accelerators in place of existing CPU-only implementations. Using the Merge dispatch system in isolation (the direct usage model), however, does not take advantage of the full extent of the Merge programming model's encapsulation and bundling capabilities, and provides little or no support for the multicore aspects of a heterogeneous multicore systems. Merge is more effective when it includes one or more high-productivity parallel languages that define independent tasks that can be mapped to different processors.

The Merge prototype provides a parallel language based on the map-reduce pattern, which underlies Google's MapReduce [15], and is present in LISP and many other functional languages. Like other specialized implementations, map-reduce implementations are encapsulated in C/C++

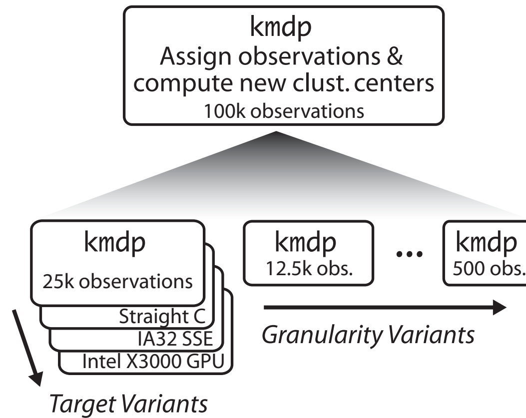


Figure 4.1: Hierarchical decomposition of k-means algorithm using functions with different granularities and target architectures.

functions and integrated alongside function-intrinsics. The map-reduce function-intrinsics provide a generic, and flexible, description of the available divide and conquer parallelism in the algorithm.

Using the map-reduce pattern, Merge applications are expressed as a hierarchical set of functions that break the computation into successively smaller operations to expose parallelism and describe data decomposition. An example is shown in Figure 4.1 for a portion of the k-means clustering algorithm. K-means iteratively determines the set of k clusters that minimize the distance between each observation and its assigned cluster. In this portion of k-means, the clusters for the next iteration are computed by averaging all of the observations assigned to a given cluster. Numerous decompositions are possible, shown as *granularity variants* in the figure, the choice of which affects the amount and granularity of parallelism. At any granularity, and at any step in the hierarchy, the application can target existing library APIs, which might already have multiple implementations for specialized accelerators, shown as *target variants* in the figure, or define new APIs that later might be extended with accelerator specific function-intrinsics. Each function call is thus potentially a choice between continuing the decomposition, or invoking a function-intrinsic that implements the parallelism in the algorithm on a specialized accelerator.

Figure 4.2 shows a sketch of the operation of Merge, and the role of the map-reduce function-intrinsics. The Merge compiler directly translates the explicit parallelism of the map-reduce pattern to independent work units, represented by the small boxes in the figure, that can be mapped to available function-intrinsics and dispatched to the processors cores. The combination of different decompositions and multiple target variants provides a rich set of potential implementations from

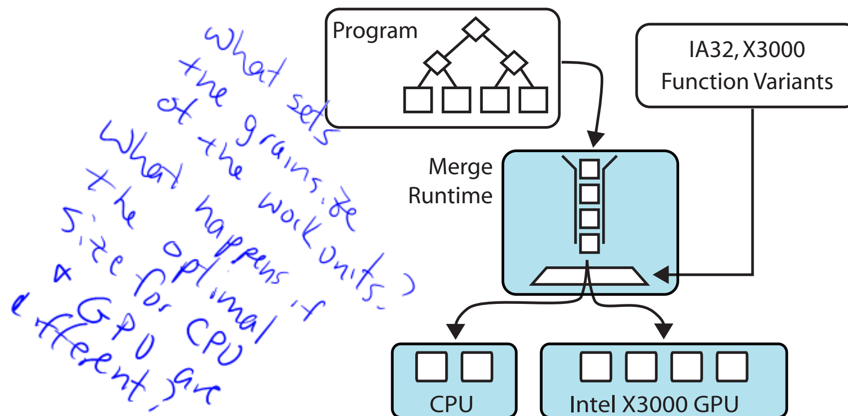


Figure 4.2: Sketch of Merge operation. The program, written using the map-reduce pattern, is transformed in work units which are pushed onto the taskqueue. Work units are dynamically dispatched from the queue, distributing the work among the processors that have applicable function variants.

which the Merge compiler and runtime can choose to optimize application performance. The following sections describe in more detail the implementation of the map-reduce language, predicate dispatch system and the runtime mapping of the map-reduce pattern to function-intrinsics.

4.1.1 Map-Reduce Parallel Pattern

Informatics applications, which often feature the integration or reduction of large datasets [17], are well described by the map-reduce pattern. In k-means clustering for example, the algorithm reduces the large number of data observations belonging to one or more discrete classes to a small set of clusters that summarize those classes.

In the map-reduce pattern, all computations are decomposed into a set of map operations and a reduce operation, with all map operations independent and potentially concurrent. The decomposition can be applied hierarchically across granularities, from single operations such as the multiplies in an inner product, to complex algorithms. An example implementation for a portion of k-means using the map-reduce pattern in is shown in lines 1-11 of Figure 4.3. The map-reduce DSL is treated as an encapsulated language, similar to the EXOCHI-based implementations targeting specialized accelerators. The different syntax (in particular the absence of surrounding `#pragmas`), is simply an artifact of the tools used to build the prototype, which are described in more detail in Section 4.2.1.

The map-reduce primitives are shown in Table 4.1. The Merge map-reduce syntax is based on the similar constructs in Sequoia [20] (along with Phoenix [57], a shared memory implementation of Google's MapReduce and generate-reducer parallelism in Fortress [2]). Unlike Sequoia, however,

Table 4.1: Merge syntax for map-reduce

`seqreduce(int i=0; i < n; i++) { ... }` Execute a function over elements of a collection sequentially, potentially performing a reduction one or more arguments. Reduction arguments are described with reduce<reducefn>(argument), indicating the function to be used in the combining. Multiple seqreduce statements can be nested, with the inner statement body only containing a function call.
`mapreduce(int i=0; i < n; i++) { ... }` Execute a function over elements of a collection in parallel, potentially performing a reduction one or more arguments. Reduction can implemented with a parallel tree.

the map-reduce statements are not an explicit description by the programmer of how a computation should be decomposed to expose parallelism. Instead, the map-reduce constructs in Merge are used as an efficient set of semantics describing the potential concurrency in an algorithm. The exact decomposition will be determined dynamically as function of the input and machine configuration.

Each mapreduce call has three parts: (1) the generator(s) to drive the map operation by creating a task tuple space, (2) the map function and (3) the optional reduce function(s). The generators work in concert with the collections, which provide data decomposition and task mapping. The collection provides the mapping between a task's tuple space and its data. A collection type does not actually contain any data, just the necessary information to provide the tuple mapping. In lines 9-11 of Figure 4.3 the function kmdp is mapped over the rows of dp and elements of assgn (indicated by the array subscript), reducing the arrays ccnew and hist produced by each map invocation. The tuple space is the row indices of dp and is created by the generator in line 9. Array2D, one of the basic collections supplied in the Merge framework, wraps a 2D matrix and is iterable by rows (similarly, Array1D wraps a vector and is iterable by elements).

The map-reduce representation is intended to capture the maximal amount of parallelism. For example, the map-reduce statement in Figure 4.3, indicates that the class assignment for each data point (the rows of dp) can be computed in parallel, the maximum parallelism at this level of the function call hierarchy. On many systems, such a small quanta of computation will be insufficient to amortize fixed overheads, however, and larger quanta of computation need to be assembled. Since each invocation of the map function must be independent (along with obeying the other restrictions enumerated in Section 3.1.2), multiple map operations can safely be coalesced. The map-reduce statement implicitly describes all potential decompositions, that is the statement in lines 9-11 of Figure 4.3, might invoke function-intrinsics with the signature

```
void kmdp(Array2D<float> dp, Array2D<float> cc, Array2D<int> assgn,
          Array2D<float> ccnew, Array1D<int> hist)
```

that operate over multiple data points, along with function-intrinsics with the signature

```
void kmdp(Array1D<float> dp, Array2D<float> cc, Array1D<int> assgn,
          Array2D<float> ccnew, Array1D<int> hist)
```

that operate over a single data point. The compiler/runtime attempts to automatically determine a smart decomposition for any particular map-reduce statement based on the function-intrinsics available. The exact selection process is described in more detail in Section 4.1.3.

4.1.2 Function Bundling

The Merge predicate-dispatch system implements the function bundling described in Section 3.1.3. The programmer identifies function-intrinsics that should be bundled together into *generic functions*, the collection of functions of the same name, number of arguments and result types [45]. The generic functions are used in place of a particular variant to provide a layer of indirection between the function caller and any particular implementation. As described in Section 3.1.3, the programmer supplies a set of annotations with each function that describe the invariants for that particular implementation. The annotations are analyzed at compile time by the bundler to produce a set of meta-wrapper functions. The syntax for a function variant and the predicate annotations is shown in Table 4.2. The meta-wrappers implement the generic function and invoke the *most-specific applicable function* (defined in more detail below) belonging to that generic function. Automatically generating the meta-wrappers enables new implementations to be integrated into a given function bundle without manually modifying pre-existing meta-wrappers, or creating new ones (similar to multiple dispatch in object-oriented languages).

The annotations, which are inspired by Roles [39], the annotation language described in [26], and Fortress [2], are of two types: predicates and groups. Predicates, which implement the input and configuration restrictions described in Section 3.1.3, take the form of logical axioms, and typically include constraints on the structure or size of inputs and the availability of particular hardware resources. For example, additional variants for `dp` are shown Figure 4.3 lines 14-45, including an X3000-specific GPU implementation. The X3000 supports 8-wide SIMD execution, so this particular variant is limited to observation vectors of length 8, and as a result of loop unrolling, four cluster centers (predicates on line 41). Groups are traits that enable hierarchical collections of variants, similar to classes with single inheritance, and are typically used to express the accelerator

```
bundle km0 {
 void kmdp(Array2D<float> dp,
           Array2D<float> cc,
           Array1D<int> assgn,
           Array2D<float> ccnew,
           Array1D<int> hist) {
  // Assign observations and compute new cluster
  // centers for next iteration in parallel
   mapreduce(int i=0; i < dp.rows; i++)
     kmdp(dp[i],cc,assgn[i],
          red<sum>(ccnew),red<sum>(hist));
}};

bundle km1 : public Seq {
 void kmdp(Array2D<float> dp,
           Array2D<float> cc,
           Array1D<int> assgn,
           Array2D<float> ccnew,
           Array1D<int> hist) {
  // Dispatch control predicate
  predicate(dp.rows < 5000);

  // ... Sequential implementation ...
  for (int i=0; i < dp.rows; i++) {
   float dist, min = FLT_MAX; int idx;
   for (int k=0; k < cc.rows; k++) {
    dist = euclidean_distance::call(dp[i],cc[k]);
    if (dist < min) { min = dist; idx = k; }
   }
   sum::call(ccnew[k],dp[i]); sum::call(hist[k],1);
}}};

bundle km2 : public X3000 {
 void kmdp(Array2D<float> dp,
           Array2D<float> cc,
           Array1D<int> assgn,
           Array2D<float> ccnew,
           Array1D<int> hist) {
    // Dispatch control predicate
    predicate(dp.rows < 5000);
    predicate(dp.cols == 8 && cc.rows == 4);
    predicate(arch & X3000_arch);

    // ... X3000 Implementation using EXOCHI ...
}};
```

Figure 4.3: Example implementation for portion of k-means algorithm; `km0::kmdp` assigns observations to a cluster and computes new cluster centers for an array of observations in parallel. `km1::kmdp` and `km2::kmdp` are variants of `kmdp` targeting traditional CPUs and the X3000 GPU.

hierarchy. For example a system may have groups for generic, sequential, SSE, and X3000 (shown on lines 14 and 33).

Similar to the implementation in [45], predicates in Merge must resolve to a Boolean and may

Table 4.2: Merge syntax for function bundling and dispatch control predicates. The literal non-terminals in the predicate syntax are constants of the specified type, while *Identifier* is any variable or field in scope.

Bundling		
`bundle dp0 : public sse { predicate(...); void dp(..); };`		
Define a new variant for `dp` with fully qualified name `dp0::dp` belonging to the group `sse` with a dispatch control predicate.		
Predicates		
pred	::=	`arch` | *lit* | *tgt* | *uop pred* | *pred bop pred*
lit	::=	*integerLiteral* | *boolLiteral* | *floatingpointLiteral*
tgt	::=	`this` | *Identifier* | *tgt.Identifier*
uop	::=	∼ | `-`
bop	::=	`&&` | `||` | `==` | `!=` | `<` | `>` | `>=` | `<=` | `-` | `+` | `*`

include literals, enumerations, accesses to arguments and fields in scope, and a set of basic arithmetic and relational operators on integers and floating point values (all arithmetic expressions must be linear). A function variant is applicable if its predicates evaluate to `true` for the actual arguments at runtime. All the variants belonging to a given generic function must be exhaustive, in that there is an applicable variant for the entire parameter space (evaluated at compile time). Exhaustiveness checking is performed in the context of groups. All of the variants in a group, as well as all variants in its parent, grandparent, *etc.*, are considered when checking for exhaustiveness.

A variant is the most specific if overrides all other applicable variants. Variant m_1 overrides m_2 if m_1 is in a subgroup of m_2 and the predicates of m_1 logically imply the predicates of m_2. Unlike more object oriented predicated dispatch systems, variants do not need to be unambiguous. As described earlier, it is assumed that many variants will be applicable, although not uniquely more specific, over a subset of the parameter space, and a secondary ordering mechanism, such as compile time order or profiling will be used to determine dispatch order. Similar to exhaustiveness checking, ordering analysis is performed in the context of groups. Variants are ordered within their group, such that any applicable variant in a group would be invoked before any variants in its parent group. In this way, high performance GPU variants on the X3000 would be invoked before defaulting to a more generic CPU-based implementation.

The variant dispatch order plays a key role in determining how an architecture-agnostic application is mapped to architecture-specific implementations. And thus the heuristic used for ordering, most-specific applicable function variant, will be a part, albeit implicitly, of any mapping algorithm. The current heuristic is motivated by the belief that on average better performance will be achieved

by invoking the function-intrinsic most specifically designed for the current input and machine configuration. The group annotation is designed to enable the programmer to influence the selection order without having to introduce unnecessary predicate annotations that don't actually correspond to actual input restrictions or other invariants.

The product of the bundler is a set of three meta-wrappers (call, lookup, and speculative lookup) for each function bundle. The call meta-wrapper simply calls the selected variant directly, while the lookup meta-wrapper returns a function pointer to the variant. Both the call and lookup meta-wrappers require that suitable variants exist and are exhaustive; speculative lookup does not have such a requirement and will support the lookup of variants that do not exist, simply returning null if no suitable variant is available. The speculative functionality enables the runtime to lookup potentially higher performing variants which may not exist, before defaulting to more general implementations.

4.1.3 Compiler and Runtime

The Merge framework compiler converts the map-reduce statements to standard C++ code which interfaces with the runtime system. Conversion consists of three steps: (1) translating the possibly nested generator statements into a multi-dimensional blocked range which will then drive execution; (2) inserting speculative function lookups to retrieve the variants for the map and reduce operations; and (3) inserting runtime calls to invoke the retrieved variants as appropriate.

The blocked range concept is drawn from the Intel Threading Building Blocks (TBB) [34] and describes a multi-dimensional iteration space that can be recursively split into progressively smaller regions. The depth of the recursion determines the extent of the parallelism, with each split creating two potentially concurrent computational tasks. The generator statements are directly mapped to a blocked range (as a result generator statements are limited to those which can be directly mapped to a contiguous range at compile time), with the blocked range serving as the index for task identification, and iteration through input and output collections.

The dataflow of a map-reduce pair forms an implicit tree with the map operations at the leaves and the reduce operations at the joins. Each leaf represents a potentially independent task that can be mapped to any of the applicable function variants. A *unit-node* function, comprising a single leaf or join, can represent too fine a granularity, however, so for many architectures there is a particular interest in *multi-node* functions, which encompass multiple leaves or joins (alternately described as encompassing a non-unit size blocked range). Invoking a multi-node variant effectively short-circuits the map-reduce structure, deferring instead to the function. Thus with a multi-node function

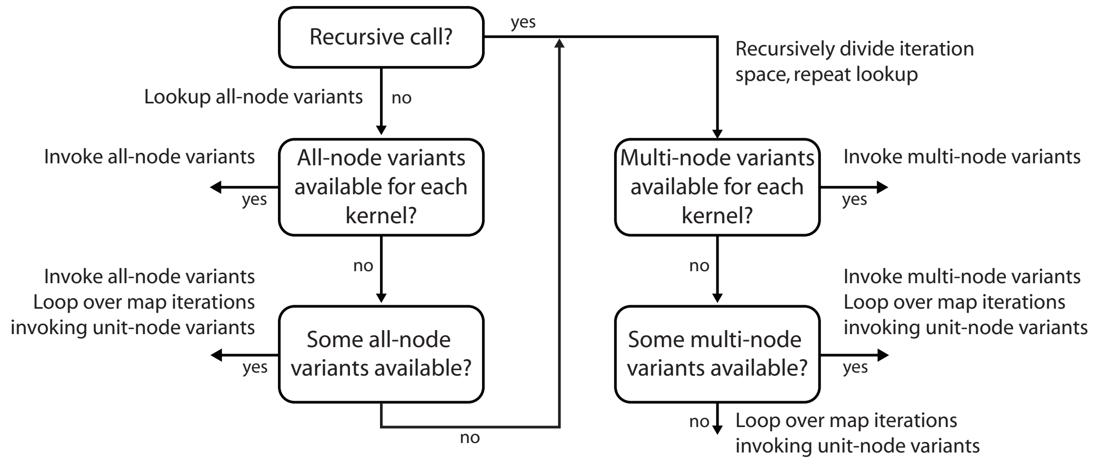


Figure 4.4: Runtime variant selection and invocation flowchart

the programmer can explicitly transform the generic map-reduce parallelism, which might otherwise be implemented as a set of unit-node tasks, to custom parallelism for a specific architecture.

Multi-node functions are preferentially selected over unit-node functions, as they potentially offer a custom parallel implementation for a specific architecture. Most of the X3000 function variants used in this thesis are intended as multi-node variants. The computation in a unit-node is often insufficient to create the necessary threads for efficient execution, while the complete function (because of reductions or other constructs) is not easily implemented completely on the GPU (*i.e.* as a unit-node variant for the parent). In these cases the variant is cast as a multi-node function that will be invoked for a portion of the map-reduce nodes.

The ordering of function lookups and resulting actions at runtime is shown in the flowchart in Figure 4.4, and discussed in detail in the following paragraphs.

The most specific translation of the generic map-reduce parallelism to a particular implementation are function-intrinsics that implement all the nodes in the map-reduce. These multi-node variants that encompass the entire map and or reduce computation are termed *all-node* variants. Based on the results of the speculative lookup for all-node variants, one of three execution scenarios is possible: all functions have all-node variants; at least one, but not all, functions have all-node variants; or no functions have all-node variants. In the first two cases, data structures to store the intermediate results are allocated if needed, and the available all-node variants are invoked directly. For the second and third case, after any all-node variants are invoked, the iteration space is recursively split into non-overlapping, potentially multi-dimensional blocks, to create independent and

potentially concurrent tasks, and similar lookup procedures repeated to identify function variants that can compute subsets of the iteration space.

The default depth of recursion is $2p$, where p is the number of processors, that is the runtime attempts to create at minimum two tasks per processor; the default can be overridden with a programmer supplied hint. The tasks are pushed onto a work queue. The Merge runtime is built on the Intel TBB parallel runtime and uses its task queue (which is itself based on the task scheduler in Cilk [8]). In short, each hardware thread keeps a “ready pool” of tasks that are ready to run. The pool is structured as a last-in first-out queue of lists. Threads execute the tasks in their queue in approximately sequential order; if their queue is empty, a thread steals tasks from the shallowest list on another randomly chosen thread’s queue.

When a task is issued from the queue, the runtime performs speculative lookups for multi-node variants for any un-executed functions in the map-reduce statement. These multi-node variants must span the entire iteration space allocated to that particular task. The same effective process is followed for the multi-node variants as for the all-node variants. Except if multi-node variants are not available for a particular function, the unit-node variant is invoked in a loop. The unit-node variant is effectively the choice of last-resort. As such there must be a unit-node available for each function, map or reduce, in the map-reduce statement; a requirement enforced by the compiler.

The per-task lookup enables dynamic workload balancing; as each processor becomes available, tasks, executing variants specific to that architecture, can be dispatched. Returning to `kmdp` in lines 1-12 of Figure 4.3 to provide an example of the lookup procedures, the map statement indicates that this function is implemented by invoking a variant of `kmdp`, which takes an observation vector, over all observations in parallel. Since `kmdp` is recursive, the runtime will directly divide the iteration space to create smaller independent tasks. If those tasks are smaller than 5000 data observations, the X3000 GPU is present and not in use, and the other input restrictions are satisfied, variant `km2` will be invoked. If the GPU is not available, or the other restrictions are not satisfied, but there are less than 5000 observations per task, variant `km1` will be invoked. If the task size is still larger than 5000 observations, variant `km0` will be invoked again to further divide the iteration space. Function variants for `sum`, not shown in Figure 4.3, will be invoked to reduce the results from the independent tasks.

The scheduling algorithm could be summarized as greedy in all aspects. The runtime greedily selects function variants that compute the largest portion of the map-reduce nodes, using the lookup

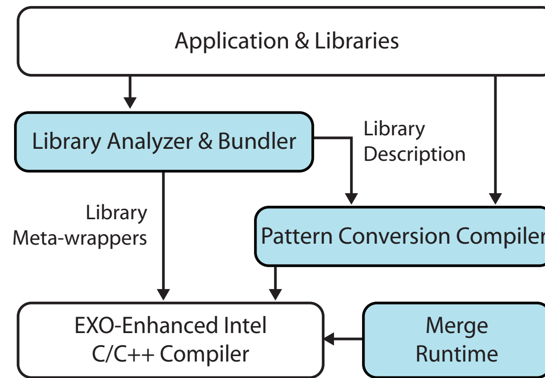


Figure 4.5: Merge compilation flow.

meta-wrappers described in Section 4.1.2, which greedily selects the most-specific applicable function for the particular input and machine configuration. As described earlier, this heuristic is motivated by the belief that on average the best performance will be achieved by translating the generic parallelism of the map-reduce statement and the generic implementation provided by a function bundle, to the most specific function intrinsic for that particular input and machine configuration.

4.2 Evaluation

A prototype of the Merge compiler and runtime has been implemented, targeting both heterogeneous and homogeneous multi-core platforms. The key characteristics of the prototype and the results achieved are summarized in the following subsections.

4.2.1 Prototype Implementation

The Merge source-to-source compiler is implemented as two independent phases: bundling and pattern conversion. Both phases are implemented using OpenC++ [11] and translate C++ with Merge keywords to standard C++ with calls into the Merge runtime. The resulting source is compiled into a single binary by an enhanced version of the Intel C++ Compiler [33] that supports EXOCHI [64]. The flow of the Merge framework is shown in Figure 4.5.

Function bundling is a global operation performed on classes identified with a bundle keyword. Function variants are bundled based on the method name, with the wrapping class serving as a unique namespace. Each variant can thus be invoked specifically through its fully qualified name

Table 4.3: Benchmark Summary

Kernel	**Data Size**	**Description**	**Porting Time**
KMEANS	k=8; 1M×8 k=8; 10M×8	k-means clustering on uniformly distributed data	3 days
BLKSCHL	10M options 100M options	Compute option price using BlackScholes algorithm	4 days
LINEAR	640×768 image 2000×2000 image	Linear image filter – output pixel is average of 9 pixel square	2.25 days
SVM	480×704 image 720×1008 image	Kernel from SVM-based face classifier	2 days
RTSS	64×3600000 samples	Euclidean distance-based classification component of neural prosthetic real-time spike sorting algorithm [70]	1.5 days
BINOMIAL	10k options	Compute options price using binomial algorithm for 500 time steps	5 days
SEPIA	640×768 image 2000×2000 image	Modify RGB values to artificially age image	1.75 days
SMITHWAT	800 base pairs 1000 base pairs	Compute scoring matrix for pair of DNA sequences using SmithWaterman algorithm	0.5 days

(and be generally backward compatible) or through the meta-wrapper. The enclosing class also provides a connection to the existing class hierarchy for group assignment. If not inherited from any class, a particular variant is assumed to belong to the generic group. If the enclosing class inherits from a class, the particular variant belongs to the group named by the parent class. At compile time the predicates are converted to logical axioms that can be used with an automated theorem prover CVC3 [6] to determine specificity and exhaustiveness (similar to the methodology and implementation in JPred [45]). In the prototype, dispatch order for ambiguous variants is determined by compile time order, relying on the programmer to order the variants as appropriate.

Pattern conversion is performed on map-reduce statements, replacing the `mapreduce` statement with code implementing the variant selection and invocation described previously. Tasks are created and inserted into the task queue with calls into the Merge runtime. The Merge runtime is built on an enhanced version of the Intel Threading Building Blocks (TBB) [34]. TBB provides a generic task driven, fork-join parallel execution runtime for shared memory platforms that has been extended to support additional dependent tasks beyond the spawning parent.

4.2.2 Evaluation Platforms

Heterogeneous: The heterogeneous system is a 2.4 GHz Intel Core 2 Duo processor and an Intel 965G Express Chipset, which contains the integrated Intel Graphics Media Accelerator X3000. The X3000 contains eight programmable, general purpose graphics media accelerator cores, called Execution Units (EU), each of which supports four hardware thread contexts. The details of the EXO implementation for the X3000 is described in [64].

The X3000 ISA is optimized for data- and thread-level parallelism and each EU supports wide operations on up to 16 elements in parallel. The X3000 ISA also features both specialized instructions for media processing and a full complement of control flow mechanisms. The EUs share access to specialized, fixed function hardware. The four thread contexts physically implemented in each EU alternate fetching through fly-weight switch-on-stall multi-threading.

Homogeneous: The homogeneous platform is a Unisys 32-way SMP system using 3.0 GHz Intel Xeon MP processors.

4.2.3 Performance Analysis

A set of informatics benchmarks was implemented with map-reduce function-intrinsics. Table 4.3 summarizes the benchmarks and inputs. All benchmarks use single precision floating point. For each benchmark, single-threaded straight C reference and Merge implementations were created. When beneficial, compute kernels are implemented on the GMA X3000 using hand coded assembly (all benchmarks, except **SMITHWAT**, had one or more kernels ported to the X3000). The porting time represents the entire time needed to create the Merge implementation, including the creation of variants for the X3000. The development of the X3000 variants consumed most of the porting time, best exemplified by **RTSS** which was relatively quick to port because it reused some X3000 variants developed for **KMEANS**.

All benchmarks are compiled with an EXO-enhanced version of the Intel C++ Compiler using processor specific optimization settings (/fast). These compiler optimizations include auto-vectorization and tuning specifically for the Intel Core 2 Duo and Intel Xeon processors in the test platforms. Performance is measured as wall clock time using the timing facilities provided by the TBB runtime.

Figure 4.6 shows the speedup achieved, relative to the straight C reference implementation, for the benchmarks using the Merge framework for one and two IA32 processor cores and the Intel GMA X3000. The Merge framework implementations show performance comparable to the

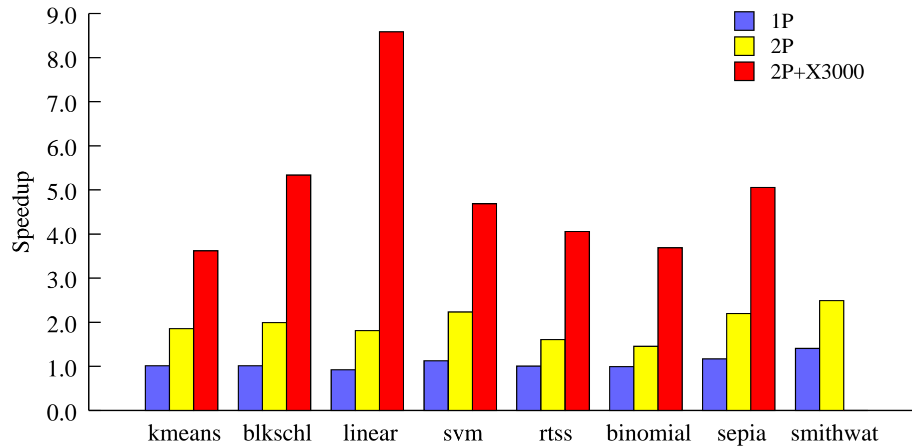


Figure 4.6: Speedup for kernels using Merge framework on 1 or 2 IA32 cores and 2 IA32 cores plus the Intel GMA X3000 vs. straight C reference implementation on a single IA32 core

straight C reference implementations executing on one processor core (.9× or better) and achieves meaningful speedups (1.45×-2.5×) executing on two processor cores. When X3000 variants are available, additional speedup (3.7x-8.5x relative to reference implementation) is obtained, exceeding what was achieved on the dual core processor alone. The parallelization for multiple cores and utilization of the X3000 are automatic and do not require any changes to the source. **SMITHWAT** utilizes a dynamic programming algorithm and is not well suited to X3000, so no variants were developed for that architecture; it is included as an example of the indirect communication capabilities of the Merge framework parallel execution model and is discussed in more detail below.

The GMA X3000 is highly optimized for image and video processing applications, and as expected shows the best performance on the purely image processing benchmarks, particularly **LINEAR** and **SEPIA**. The other benchmarks are not so readily mapped to the X3000's image-oriented SPMD execution model, in particular its SIMD arithmetic unit and the 2-D block-optimized memory system, and thus do not achieve the same kind of speedups on the X3000. However, benchmarks with similar CPU and X3000 performance can often benefit from cooperative multi-threading, in which the X3000 and IA32 CPU concurrently process disjoint subsets of the map operations. In these cases the X3000 functions less as a distinct accelerator and more as an additional processor core.

Figure 4.8 shows the speedup achieved using different cooperative multi-threading strategies. The activity of the processors under the different strategies is sketched in Figure 4.7. When the accelerator code is invoked using EXOCHI, the CPU thread manages the accelerator, waiting until

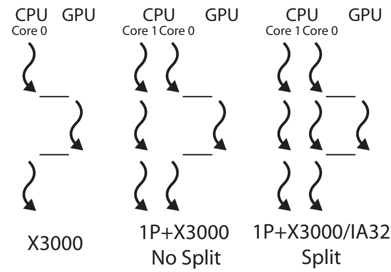


Figure 4.7: Sketch of thread execution under different cooperative multi-threading scenarios

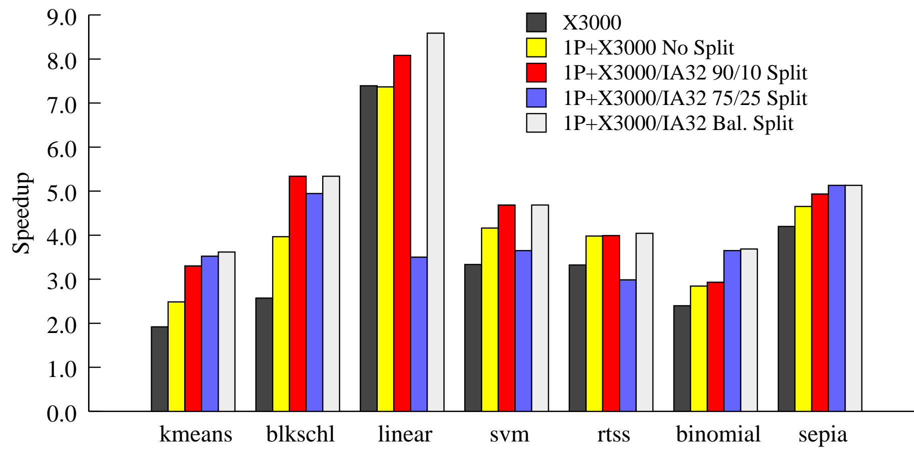


Figure 4.8: Speedup for kernels using Merge framework on heterogeneous system using different cooperative multi-threading approaches between the Intel GMA X3000 and IA32 CPU

all of the accelerator threads have completed before proceeding. Thus if only one CPU core is used (X3000 in Figure 4.7 and Figure 4.8), the CPU and X3000 do not execute concurrently, and any speedup results entirely from using the X3000. This is the level of performance that might be expected from a system that focuses only on compiling general purpose code to specialized accelerators.

When multiple CPU cores are used (1P+X3000/IA32 No Split), the Merge framework automatically distributes work between the accelerator and the second CPU core. In this regime, the runtime first attempts to invoke the X3000 variant. If the X3000 is busy, the runtime defaults to the next best variant, which in this case targets the CPU. For the less image oriented benchmarks, such as **KMEANS** and **BLKSCHLS**, where the CPU performs well relative to the X3000, the automatic cooperative multi-threading can significantly improve performance. Performance increases over X3000 of 28% and 54% were obtained for **KMEANS** and **BLKSCHLS** respectively.

In the first two regimes, the CPU thread which serves as the manager for the accelerator threads idles until the accelerator completes. EXOCHI supports limited nowait fork-join, in which the CPU master thread can continue to perform useful work before blocking at the join point. Ideally, after forking the accelerator threads, the managing CPU could return control to the runtime and participate in dynamic task allocation. However, limitations in the prototype implementation of EXOCHI, particularly in how the X3000 signaled completion, prevented this. Instead, the working set of the variant is statically partitioned between the CPU and GPU (1P+X3000/IA32 x Split, where x is 75/25, 90/10 and balanced). Function-intrinsics designed to split the work are in effect targeting a combination CPU and GPU hardware resource (an example of the more complex resource sets described in Chapter 3).

Three different partitioning schemes are shown in Figure 4.8: (1) a 90% CPU, 10% GPU split; (2) a 75% GPU, 25% CPU split; and (3) a balanced split where the partitioning was selected so the GPU and CPU finish as close the same time as possible. It is this last scenario that is presented as the X3000 bar in Figure 4.6. Most benchmarks, excluding **RTSS**, benefit from being able to use all the available CPU cores and the GPU. For **KMEANS** and **BLKSCHLS** in particular, improvements of 45% and 34% respectively, were obtained over the 1P+X3000/IA32 No Split regime. In general those benchmarks for which CPU and X3000 performance was comparable benefit from a more equitable work partitioning. **RTSS** does not show any improvement because the small reductions in the workload sent to the X3000 does not reduce the overall execution time, instead some execution units are simply idle. Extending the EXOCHI and Merge prototypes to support full dynamic scheduling of the manger CPU thread is an area of ongoing work. Note that this static partitioning does not affect the dynamic scheduling of the other cores or accelerators.

To estimate the execution time breakdown, we instrumented the different function-intrinsics (CPU and GPU) that are invoked in each benchmark running under the 1P+X3000/IA32 Balanced Split regime[1]. For those benchmarks where most or all of the computation can be ported to the X3000, such as **BLKSCHLS**, **LINEAR**, **SVM**, **RTSS** and **SEPIA**, data was being transfered to X3000 or the X3000 was directly computing more than 75% of the time. Relatively few functions are invoked in these benchmarks, and so the overhead of the Merge predicate dispatch system is negligible. Other benchmarks, namely **KMEANS** and **BINOMIAL**, the X3000 is only active ∼40-50% of the total execution time. Compared to the other kernels, a smaller portion over the overall computation can be ported to the X3000. More significantly though, there was a mismatch between the work

[1] Since many of the function-intrinsics execute for only a short time, however, the measurements are imprecise, and so we restrict ourselves to a discussion of the broader trends

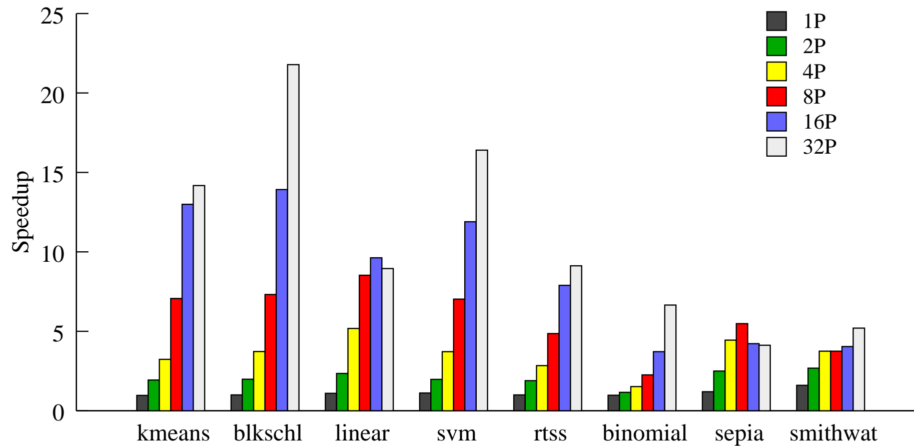


Figure 4.9: Speedup for kernels using Merge on the homogeneous 32-way SMP machine vs. straight C reference implementation on a single core

blocks created by the runtime, and those that can be processed by the X3000. The combination of limited capacity for queuing X3000 threads and the minimum workload size required to amortize the data transfers restricts the use of the X3000 to a limited range of input sizes. After processing as much of the work unit as is productive on the X3000, the remainder is processed on the CPU. In the cases of **KMEANS** and **BINOMIAL**, these remainders add up to non-trivial amount of computation. Better utilization could be achieved by creating function-intrinsics that manually control scheduling over larger portion of the input dataset (implemented as a new variant), although at the cost of more programmer effort, or by optimizing the division of the input dataset into work units that are efficient for the X3000. Extending Merge to support adaptive or profile-driven workload partitioning is an interesting topic for future research.

Figure 4.9 shows the speedup relative to the straight C reference implementation achieved on the 32-way SMP homogeneous test platform for the same benchmarks (with the larger input datasets). As with the heterogeneous platform, using the 32-way SMP system did not require any changes to the source.

The CPU-only benchmark **SMITHWAT** takes advantage of the indirect communication provided by the Merge framework parallel execution model. The benchmark implements the scoring matrix generation portion of the Smith-Waterman dynamic programming (DP)-based DNA sequence alignment algorithm. The DP algorithm creates dataflow dependencies between the otherwise independent entries in the score matrix, which in a parallel model that only supports direct communication through task call and return is cumbersome to express. Using indirect communication via

the collection arguments, the task for a matrix entry can register its dependencies and switch-out until the required entries have been computed, allowing the Smith-Waterman algorithm to be effectively parallelized without resorting to highly specialized implementations or DP-specific parallel patterns.

When highly optimized parallel implementations are available for an algorithm, such as for IA32 SSE, or using existing low-level thread constructs, they can be incorporated into the library and invoked directly. Thus in general, the performance of the Merge framework's library-based approach will track that of the best available implementations, ensuring that even in the worst case the Merge framework implementation will perform comparably to existing implementations.

Note that the X3000 used in the heterogeneous test platform is a chipset integrated GPU and does not have the computational resources or memory bandwidth of a discrete GPU, which limits performance in absolute terms. However as the real speedups shown here indicate, the X3000 can provide useful acceleration, especially when used with the CPU as part of a whole system approach. Further, many users will already have an X3000 or similar GPU available in their systems, offering additional speedup relative to the CPU alone for no additional hardware cost. And as described earlier, the Merge framework can be used with any accelerator that can support the EXO interface, including discrete GPUs.

Chapter 5

Heterogeneous Microarchitectural Extensions

The introduction of SIMD extensions to commodity x86 processors and the development of application-specific instruction set processors (ASIPs) have made tightly-coupled heterogeneous systems widely available. However, as described in Chapter 2, these tightly-coupled systems inherit many of the architectural limitations of their progenitors, including restricted instruction execution semantics, inefficient memory systems and limited or inefficient mechanisms for inter-core interaction. This chapter presents a chip-multiprocessor architecture (CMP) with a modular architecture that enables the integration of non-trivial extension modules into traditional general-purpose processor (GPP) cores. These extensions enhance legacy architectures with asynchronous bulk memory accesses, low-latency inter-core interconnect and atomic interaction features typically only available in specialized discrete accelerators, such as GPUs or the Cell processor.

Much like discrete accelerators, the extended system offers functionality that significantly differs from the base CMP, and for which there is often limited or no significant high-level compiler support. The availability of many different combinations of extensions make portable programming a further challenge. The Merge programming model enables these heterogeneous microarchitectures by supporting the integration of multiple different extension-specific compiler toolchains. Programmers can create specialized function-intrinsics, using encapsulated extension-specific assembly or domain-specific languages, that are bundled alongside other implementations targeting different input or machine configurations. The capability to provide multiple implementations for the same computation, including legacy implementations, or those that emulate extension features, ensures that applications can be made backward and forward-compatible without compromising

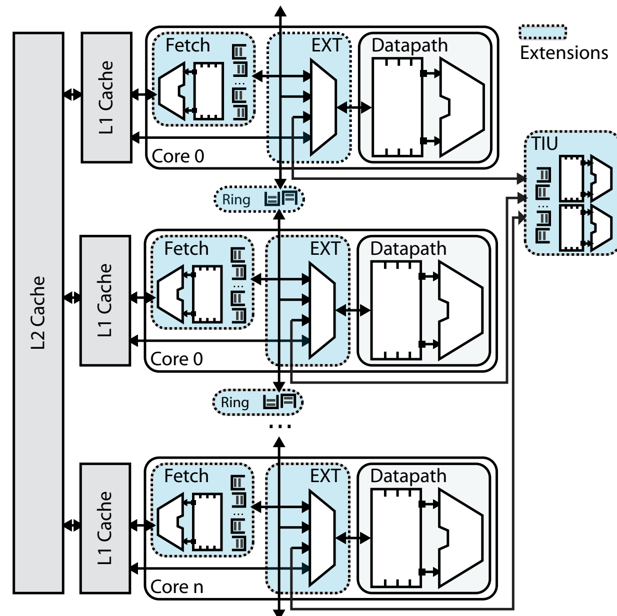


Figure 5.1: Block diagram of extended CMP architecture. Extensions are highlighted with a dotted outline and light blue background.

performance on either legacy or extended processors.

This chapter describes the modular microarchitectural interface, introduces three extension units, describes how Merge is used with the extended CMP, and presents an detailed performance evaluation of the extended CMP architecture using a cycle-realistic execution-driven simulator.

5.1 Modular Microarchitectures

Modular microarchitectures extend traditional designs by integrating specialized extensions into and among traditional GPP cores. Figure 5.1 shows the block diagram of an example CMP with three extensions: a software-controlled data fetch engine (Fetch), a thread-interaction unit (TIU), and a ring-based inter-core network (Ring); new hardware modules are highlighted with a dotted outline and light blue background.

5.1.1 Extension Interfaces

An extension's interface is mapped into a restricted segment of the virtual address space, and accessed with load/store instructions. Since the extensions are memory-mapped, no new instructions

need to be added to the ISA to support the new modules. Designers can reuse most of the existing processor core, including the cache memory system, instruction decoder and processor datapath, without change. The extension interface is designed to minimize the changes to the processor datapath and instruction decode while imposing the fewest restrictions on the interface to or construction of the extensions. The architecture described in this thesis uses a memory-mapped interface for those reasons. However, almost every aspect of this work is equally applicable for an extension interface that uses extension write and read instructions in place of memory mapping.

A key distinction between the memory mapped interface and similar coprocessor interfaces, such as those provided by the MIPS ISA [55], is how commands and data are transferred to the extension. With the memory-mapped interface, the command, alternately the extension "instruction", is encoded in the address of the memory operation. The instruction word is not hard-coded in the instruction stream, instead it can be generated at runtime using any of the addressing modes provided in the base ISA. The MIPS ISA, in contrast, provides separate instructions for transferring data between the processor and coprocessor, and initiates coprocessor execution with a statically-coded coprocessor instruction, `COPz`. Although the same extensions could be implemented with either interface (subject to the limit of two coprocessors for MIPS), MIPS would require more base processor instructions to accomplish the same task.

Each extension unit defines a set of physical ports, specific to operations it performs, that are directly mapped into the virtual memory space. With these ports, data and or instructions can be passed between the CPU core and the extension.

Stateful extensions create challenges similar to memory-mapped I/O for architectures with speculative instruction issue. Any given access to an extension might result in changes in state, and thus loads (or extension access instructions) are no longer idempotent. This work builds on base CMP with in-order cores and no speculative memory accesses (similar to Sun's Niagara [37] architecture). Since future many-core systems are likely to use in-order cores [3], using an in-order base core is a reasonable assumption. Modular microarchitectures and stateful extensions can be used with out-of-order (OoO) designs either by blocking speculative access to the extension portion of the address space, or introducing speculation buffers and rollback logic into the extension interfaces (similar to the speculative queue interfaces in [40]).

5.1.2 Extender Unit

Specialized extensions are connected to the base processor through the *Extender* unit (EXT). The EXT is integrated into the processor's load/store (LS) unit, and is responsible for demultiplexing

memory accesses to either the memory system or one of the extension units. Memory requests are processed in two steps: (1) requests to the user segment of the memory space are routed to the base processor's LS unit, while requests to the extension segment are processed by EXT; and (2) requests made to an address that maps to an extension interface port are translated to the format expected by that port (typically dropping the high-order bits) and forwarded directly to the extension along with the requester's process ID (PID).

Memory protection and virtualization is the responsibility of the extension. Since the extensions can be so different there is unlikely to be one "best" mechanism, implemented in the EXT unit, that can be shared among them. Instead each extension implements their own protection mechanism, using the PID that is forwarded along with each request. When an application initiates an extension, the OS will allocate the necessary resources within the extension, marking them as appropriate with the owner's PID. Subsequent requests will be validated against that PID in the way most appropriate for that extension. If an access violation (or some other type of exception) occurs, the exception is resolved through Collaborative Exception Handling (CEH) [64] with the triggering CPU. When the exception occurs, the extension interrupts the CPU that originated the request, and invokes a handler to process the exception. CEH is used for all exceptions originated by the extension units.

5.2 Extension Units

The design of the example extension units are motivated by the structure of information processing (informatics) applications. Informatics applications typically analyze, mine or synthesize the growing digital data corpus. These actions all share an asymmetric bandwidth profile: analysis algorithms consume most or all of the dataset to produce a small model that describes in the input dataset; data mining scans large amounts of data to identify a few examples of interest; while synthesis produces a large dataset or synthetic world from a compact model. The input-output bandwidth asymmetry indicates that extreme data reduction or generation will an important part of all three classes. The example extension units are designed to improve performance of informatics applications by accelerating bulk memory accesses (Fetch Unit) and providing efficient reduction/-generation operations (Ring Network, Thread Interaction Unit).

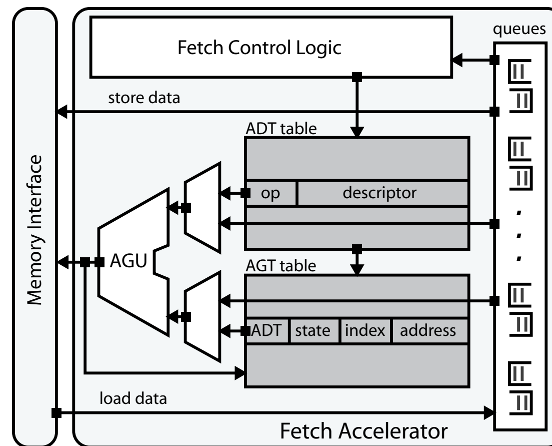


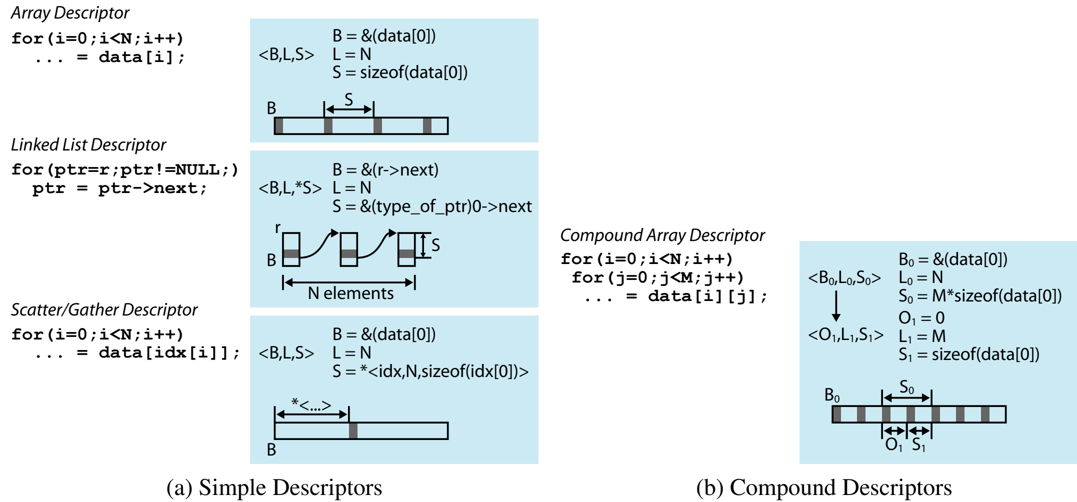
Figure 5.2: Block diagram of Fetch extension

5.2.1 Fetch Unit

Numerous applications operate on well structured data sets; a decoupled data fetch engine can exploit this structure to hide memory latency. The Fetch unit provides decoupled strided, segmented accesses, indirect scatter/gather and simple pointer chasing. Based on Choi's multi-chain prefetcher [12], the Fetch unit extends that design by supporting actual accesses as well as prefetching, and providing indirect scatter/gather. The interested reader is pointed to [12] for more details.

A block diagram for the fetch unit is shown in Figure 5.2. The interface provides four physical port types (not explicitly shown in Figure 5.2): (1) a control port; (2) control data ports; (3) load data ports and (4) store data ports. The processor sends commands, such as load access parameters into internal memory structures, to the fetch unit via the control port, including any necessary control data on the control data ports. The load and store data ports are used for the actual (that is non-control) memory accesses and are backed by queues to provide decoupled loads and stores.

The core of the fetch unit are the *access descriptor table* (ADT) and the *address generator table* (AGT). The ADT contains the static parameters that describe a structured access, such as base address, stride, and transfer queue, while the AGT maintains the current state of an access, such as the current address and index. The values of ADT are supplied by the processor and do not change, while the data in an AGT entry is updated after each access. Initiating a structured access allocates and populates an AGT entry using the parameters from the ADT. On each fetch cycle, a valid AGT entry is selected via round-robin arbiter, the appropriate memory request is generated and pushed to the data cache, and the AGT entry is updated with the address for the next iteration.



(a) Simple Descriptors (b) Compound Descriptors

Figure 5.3: Example simple and complex fetch descriptors, adapted from [12]

Structured access are assembled from a composable set of simple strided or pointer chasing accesses, expressed using LDS descriptors (adapted from [12]). Basic descriptor types are shown in Figure 5.3. Simple array descriptors contain three parameters: base (B), length (L) and stride (S) along with an optional data queue (Q) for non-prefetch accesses. A simple linked-list traversal is similar, although the stride parameter specifies the offset of the "next" pointer from the base, and at each fetch cycle, the base is updated with the "next" pointer. Indirect scatter/gather (SG) accesses are also similar, albeit obtaining the stride from one of the data queues. SG accesses require a separate access descriptor for the index stream.

The simple accesses described above can be composed to create more complex access patterns. An example compound access for a two-dimensional array is shown in Figure 5.3. Other example compound accesses would be blocked arrays, arrays of lists, and other combinations of the three simple access types. When an ADT entry is a parent, *i.e.* not the lowest level of the compound access, instead of generating a memory request, it will allocate and initialize a new AGT entry during the fetch cycle. Depending the access mode, discussed in more detail below, the parent may not be available for selection until its (potentially compound) child completes.

When data is actually obtained from the cache (as opposed to simply prefetched into the L1 cache) accesses are single-chain, meaning the address stream is generated in-order. In this case, a parent will not be eligible to generate a request until its child is complete. When an address stream is only being used for prefetching, requests can be issued out-of-order, or multi-chain. In multi-chain mode, each level of a compound access will generate requests as quickly as allowed by the

scheduling logic. Consider the example compound request from Figure 5.3, in single-chain mode, all the requests for a given row will be generated before the next row is begun. In contrast, in multi-chain mode, a new AGT entry can be created for row $i + 1$ while requests are still being generated for row i.

5.2.2 Ring Network

Loops with loop-carried dependencies can be difficult or expensive to parallelize on architectures without low cost mechanisms for fine-grain blocking communication between threads. The bidirectional Ring Network extension module, shown in Figure 5.1 provides low-overhead blocking messaging between adjacent processor cores. The Ring module is motivated by the similar functionality in the SCALE architecture [38]. The simplest of the extension modules, the Ring network unit contains two blocking FIFOs, and ports for each processor to read from and write to the queues.

5.2.3 Thread Interaction Unit

The thread interaction unit (TIU) enables low overhead global interaction between threads executing on different processor cores. The TIU provides a set of global accumulation buffers that are shared among all of the processor cores, and a compact ISA that operates on those buffers. A block diagram for the TIU is shown in Figure 5.4. Common uses for the TIU are global accumulators, such as might be used in a histogram operation, synchronization primitives, such as barriers or locks, and associative operators.

The interface provides three physical ports for each core: (1) a control port; (2) two input data ports; and (3) an output data port. The processor sends commands, such as add to accumulator via the control port. Commands take the form of an $\langle op, key \rangle$ pair, which indicates that operation op should be performed on the accumulator indexed with key. Data is passed between the TIU and processor via the input and output data ports. Like the other extension modules, the physical ports are backed by queues to decouple execution. The queues are local to the processor cores. On each TIU cycle a round-robin arbiter selects a command, data pair to forward from the processor to the TIU, or a data result to forward from the TIU to the processor.

The core of the TIU is two-way VLIW processor and a register file that stores the accumulator buffers. Each VLIW instruction includes control bits for selecting entries from the register file, forwarding the results to the ALUs, indicating the ALU operation, and forwarding the results back to the register file or the processor cores. Command operations received from the processor cores

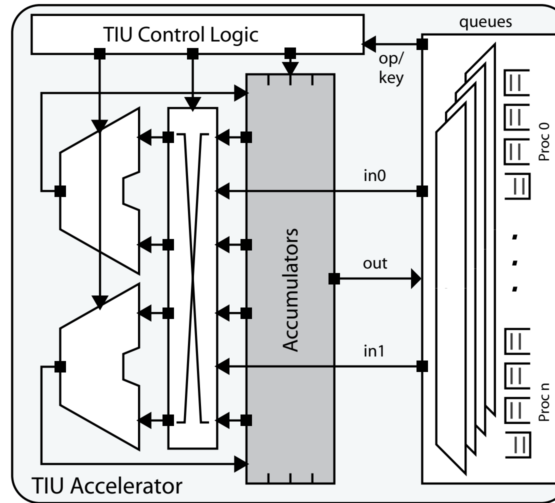


Figure 5.4: Block diagram of the Thread Interaction Unit extension

are translated to an internal VLIW instruction for execution by the TIU processor. In addition to the common operations that are predefined in the TIU interface, programmers can specify their own sequence of internal operations that are executed upon receiving the appropriate external command.

5.3 Programming Model

The programming model for heterogeneous microarchitectures builds on the system-oriented techniques implemented by EXOCHI [64] and the Merge programming model. In particular, the programming model addresses two key challenges: (1) integrate extension-specific assembly or domain-specific languages into traditional C/C++ code, (2) ensuring the correct and best code is invoked when extension modules may or may not be present, might be busy, or otherwise unavailable. Architectural and programming mechanisms from EXOCHI are implemented to support the integration of extension-specific code into C/C++ code, and support cooperative execution between the extension modules and the GPP cores. The dynamic dispatch system from the Merge programming model is used to control code invocation and emulation, responsive to the dynamic state of the system. The compilation flow for the extended systems is shown in Fig. 5.5 and will be discussed in more detail in the following sections.

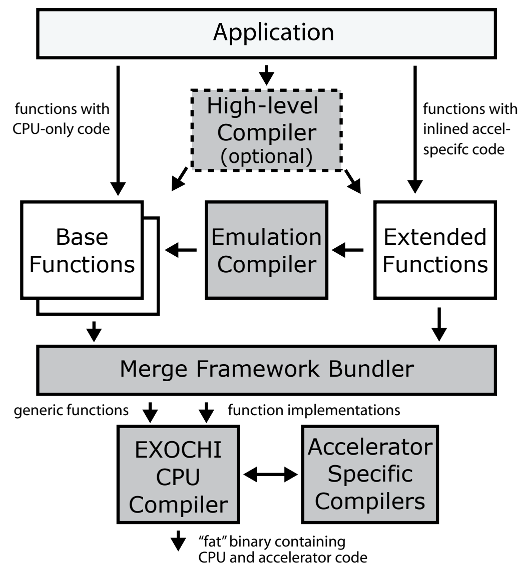


Figure 5.5: Compilation flow

```
#pragma extend {
  queue<int> dQ;  // data interface
  // local bucket accumulation
  #pragma section target(CPU) in(dQ) {
    for (int i=0; i < N; i++)
      loc_bkt[key(dataQ)]++;
  }
  #pragma section target(fetch) out(dQ) {
    dataQ = <&data[0],N,sizeof(int)>;
  }
}
```

Figure 5.6: Sample code for histogram

5.3.1 Execution Model

The extension units execute concurrently and independently from the GPP cores. Like the GPP cores, extensions maintain architecturally meaningful state. However, unlike the CPU cores, this state is not directly visible to the OS. Instead operations, such as exceptions, that require OS intervention are proxied through the appropriate GPP core via interrupt handlers. Only module-independent structures are directly manipulated by the OS. Providing a level of indirection between the OS and the extension modules minimizes the amount of knowledge about any given extension the OS must have, enabling extensive reuse of the existing software infrastructure while encouraging rapid evolution in the extension space.

Although the extensions provide protection and virtualization mechanisms, these capabilities are not, and are not intended to be, as robust as those offered by GPPs. Many extensions, such as the Ring network, derive their efficiency benefits from bypassing traditional abstractions, such as stand alone cores, to provide more tightly coupled systems. Further, many extensions have limited internal memory or other similar usage restrictions, that cannot easily be virtualized. Much like architectural registers in current architectures, the extensions represent separate memory spaces that must be explicitly managed by the programmer/compiler. As such not all extensions are compatible with the cores being used in a stand alone fashion, such as in traditional multi-core processors.

For those extensions, like the Ring network, that offer inter-core interaction and are not easily virtualized, the entire processor (or the relevant subset) is treated as a single resource, using the concept of flexible resources sets described in Section 3.3. In such a scenario the applications threads are not mapped to cores by the OS, but instead the application is scheduled onto the entire system, and its threads are allocated as appropriate by the application. With this capability, applications can explicitly schedule communication, allocate finite software-controlled memory resources, or perform other actions that could be incompatible with pre-emption of threads.

5.3.2 Integrating Extension-specific Code

The programming front-end for heterogeneous microarchitectures is motivated by EXOCHI [64], which was described in detail in Section 3.1.1. Code targeting an extension module is contained inside a specialized pragma which defines the scope of the extension usage. An example implementation for histogram is shown in Figure 5.6. Each extended section in the program contains three subsections: (1) a preamble declaring interface data structures; (2) the CPU or (CPU/other extension)-targeted code; (3) extension-specific assembly or domain-specific language targeting the extension module. When non-CPU code is encountered, the EXOCHI compiler invokes a separate extension-specific compiler (or assembler) to produce the necessary sequence of memory accesses to control the extension, and integrates them with the CPU-targeted code (as shown in the bottom of Figure 5.5).

The interface data structures provide the link between the extension module and the processor core. Although the interface structure appears to be memory-based data structure, a FIFO in the example in Figure 5.6, and could be implemented as such, it is intended to be mapped to dedicated hardware resources provided by the extension module. A limited set of structures is available, consisting of those that can be readily mapped to extension features, such as the port queue. The preamble section which contains interface structure declarations may also contain configuration calls or other commands required to initialize the extension modules.

The CPU-targeted section is marked with a `target(CPU)` directive along with additional directives indicating any interface data structures, and their type. In the example in Figure 5.6, the histogram computation reads data produced by a structured fetch operation. During compilation the read from the interface queue is converted to the appropriate memory request, in this case a load from the address mapped to one of the fetch extension's data output queues. The extension-targeted section is marked with a similar `target` statement indicating the extension, along with directives

for the interface data structures. Inside the pragma, the extension actions are encoded with assembly or a domain specific language. In the example, the array access is expressed using the <B,L,S> notation (described in Section 5.2.1); a contiguous array, starting at &data[0], of length N and stride sizeof(int), is fetched.

5.3.3 Backward/inter compatibility

Since the extension modules function as "add-ons", the heterogeneous microarchitectures can execute legacy code without modification. Applications written to take advantage of an extension(s), will not, however, be backward compatible with legacy systems, or necessarily inter-compatible with extended systems that have different configurations. Traditionally, software emulation has been used to provide backward/inter-compatibility among heterogeneous hardware, but as will be shown in Section 5.4 the performance penalty for emulation can be significant. Instead of relying exclusively on emulation, and thus potentially being forced to accept worse performance for new code on legacy hardware, we present multi-level solution for managing system compatibility, shown in the top of Figure 5.5 and built on the code selection techniques in the Merge programming model, that ensures little or no slowdown relative to the baseline implementation targeting the base CMP alone.

When an extension provides functionality with no real counterpart in the base CMP, such as the blocking Ring network, the emulation overhead can result in worse performance than the baseline implementation. In these cases emulation alone is not an effective solution to compatibility, and additional base CMP-targeted implementations, using different algorithms might be needed. Using Merge, an application (or library or compiler) can provide multiple implementations for the same computation, some targeting the base CMP and others the extended CMP; the choice among which enables backward and inter-compatibility without unnecessary emulation overheads.

The different function implementations can be: (1) created by hand by the programmer; or (2) generated by higher-level compilers, *e.g.* a vector compiler, that can target the specialized extensions. Along with base CMP-only implementations provided by programmer (or created by the high-level compiler), a separate emulation compiler can automatically translate extension-specific code into a base CMP-only implementation or implementations that target some subset of the extensions. These implementations are bundled together using the techniques described in Chapters 3 and 4 to create a set of meta-wrappers that provide a level of indirection between the application and the particular application. As described previously, applications target the meta-wrappers, and

the runtime system automatically and transparently maps the function call to a particular implementation based on the annotations, the configuration of the machine, and the relative performance of the different variants (as determined by performance modeling or profiling).

Merge-based code selection enables both “bottom-up” and “top-down” approaches to portable programming. In the bottom-up approach, the programmer can start with an existing code base, and augment that implementation with new function variants that exploit the microarchitectural extensions. The new variants are automatically “bundled”, as described above, and invoked when appropriate. Since the programmer began with a base CMP-only implementation, the application is backward compatible from the start, while also capable of transparently targeting the extensions through the addition of new function variants. In the top-down approach, a high-level compiler generates variants for the different system-configurations, including a base CMP-only implementation, and uses Merge to select among them as appropriate for a particular system configuration.

The combination of function bundling and performance-guided code selection ensures that performance is never worse than the baseline implementation. Even when implementations are generated by a compiler, including the emulation compiler, the programmer is always able to augment the variants with an optimized base CMP-only implementation that will supersede worse-performing variants and serve as the performance baseline. Similarly the programmer is not dependent on the availability of a high-level compiler to be able to target rapidly evolving extensions; new variants can be created that directly invoke extension features, and automatically bundled into the application alongside compiler-created implementations.

5.4 Evaluation

An example extended chip-multiprocessor architecture (CMP) has been implemented. The key characteristics of the architecture and the performance results are summarized in the following subsections.

5.4.1 Experimental System

A detailed cycle-level, execution driven microarchitectural simulator has been developed (based on the Sim3 simulator [52]). Details modeled during simulation include integer and floating point execution pipelines, data and instruction L1 caches, L1 to L2 crossbar, multi-bank L2 cache, the extender unit and the Fetch, Ring and TIU extensions.

Table 5.1: Simulation parameters

Base CMP	
Cores	8
Datapath	7-stage single-issue in-order
L1 I-Cache	16 kB, 8-way, Random replacement
	32 b line, WrT, non-WrA, 2 cycle latency
L1 D-Cache	16 kB, 4-way, LRU replacement
	16 b line, WrT, non-WrA, 2 cycle latency
L2 Cache	4 MB, 8-banks, directory coherence
	16 way, 64 b line, 12 cycle latency
Main Memory	100 cycle latency
Fetch Accelerator	
Queues	8 in, 8 out, 8 entries deep
AGT/ADT	64 entries
TIU Accelerator	
Queues	8 entries deep
Accumulators	64×4 words
Instructions	16 entries
Ring Accelerator	
Queues	8 entries deep

The base CMP architecture is loosely based on the Sun Niagara processors [37], (without simultaneous multi-threading) using the SimpleScalar PISA ISA [4]. The default simulation parameters are shown in Table 5.1.

Applications are compiled using gcc (-O3), with the extension-specific assembly converted to appropriate load/store sequences by a separate prototype front-end. A series of benchmarks have been implemented, and augmented with extension-specific code as appropriate. Table 5.2 summarizes the benchmarks and inputs along with the particular extensions used. Reported performance is based on the cycle count, not including the start-up periods of the benchmark.

5.4.2 Comparing Base and Extended CMPs

Figures 5.7 and 5.8 show the relative execution time (normalized to the base architecture with one core) for the benchmarks on both the base and extended CMPs. The execution time is broken

Table 5.2: Benchmark Summary

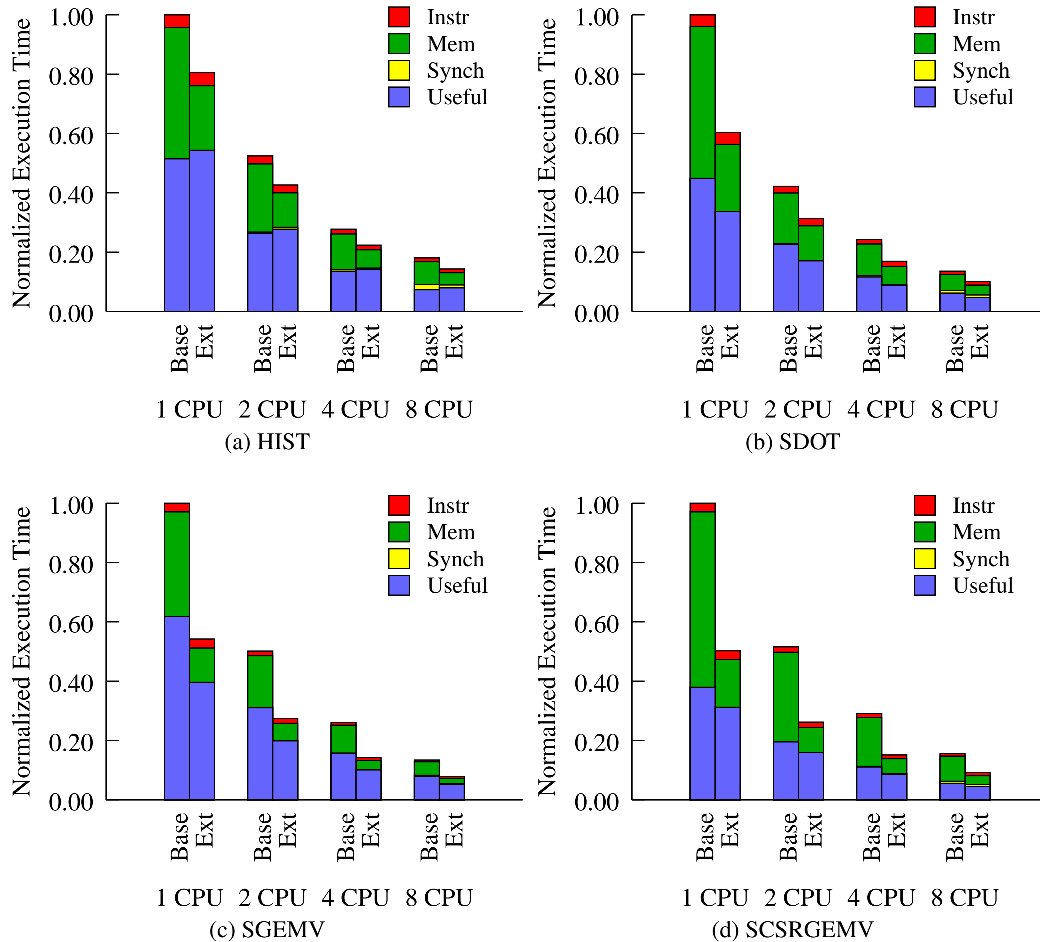
Kernel	**Data Size**	**Description**	**Exten**
HIST	n=32768	Histogram of uniformly distributed integers into 32 buckets	Fetch, TIU
SDOT	n=16384	Single precision BLAS 1 dot product	Fetch, TIU
SGEMV	[256×256]	Single precision BLAS 2 matrix-vector multiply	Fetch
SCSRGEMV	[1030×1030]; 6.65 nz/row	Single precision BLAS 2 sparse matrix-vector multiply	Fetch
SGEMM	[128×128]	Single precision BLAS 3 matrix multiply	Fetch
KMEANS	k=8; 1024x8	k-means clustering on uniformly distributed data	Fetch, TIU
SMITHWAT	512 base pairs	Compute Smith-Waterman scoring matrix for DNA sequences	Ring

out into time spent performing useful work (useful), waiting locks or other forms of synchronization (synch), waiting on memory accesses (mem) and waiting on instruction access (instr). For all benchmarks, and all configurations of cores (1,2,4 and 8 cores) the extended architecture achieves improved performance compared to the base design.

The benchmarks that are memory bandwidth limited, and have well structured accesses, such as the BLAS kernels and **KMEANS**, that can take advantage of the Fetch extension show the largest performance improvement. In these cases the decoupled DMA-like memory accesses ensure that bandwidth is maximized by aggressively generating requests, while simultaneously reducing the address computations and other bookkeeping that must be performed by the main CPU. These combined effects are very pronounced for **SGEMM** which is memory bandwidth limited and has a non-trivial array access pattern.

good prefetcher would do well for these apps too, right?

Figure 5.9 shows the number of instructions and distribution of instruction types executed on 8 core base and extended CMPs. As expected the benchmarks with more complex array access patterns achieve the largest reduction in instructions executed. These benchmarks show a corresponding improvement in energy efficiency as operations that are typically performed on the low-efficiency CPU and transferred to the higher-efficiency extensions. To quantify the improvement, the simulator was augmented with activity-driven energy estimation of the major memory and datapath structures using data from [5], CACTI 4.2 [61], and circuit-level estimation. The extended



(a) HIST
(b) SDOT
(c) SGEMV
(d) SCSRGEMV

Figure 5.7: Execution time of benchmarks on both base and extended CMPs. Time normalized to the base CMP with a single processor core.

CMP consumes between 8% - 49% less energy than the base design (geometric mean = 28%), with the improvement for each benchmark closely tracking the reduction in instructions executed.

L1I-cache accesses are the largest energy consumer in the experimental system (instruction fetch consumes 60%-80% of the total energy, in line with results in [5]). Thus from an energy efficiency perspective, one of major goals of the extensions is to eliminate instruction fetches. In the example system, this is accomplished by moving address calculations into the Fetch unit, and eliminating lock-polling by using TIU accumulators. In addition to reducing the overall number of instructions, the extended CMP also executes relatively fewer legacy instructions, replacing them with accesses to the extensions (indicated by `Ext` class in Figure 5.9). Although in the default extended CMP,

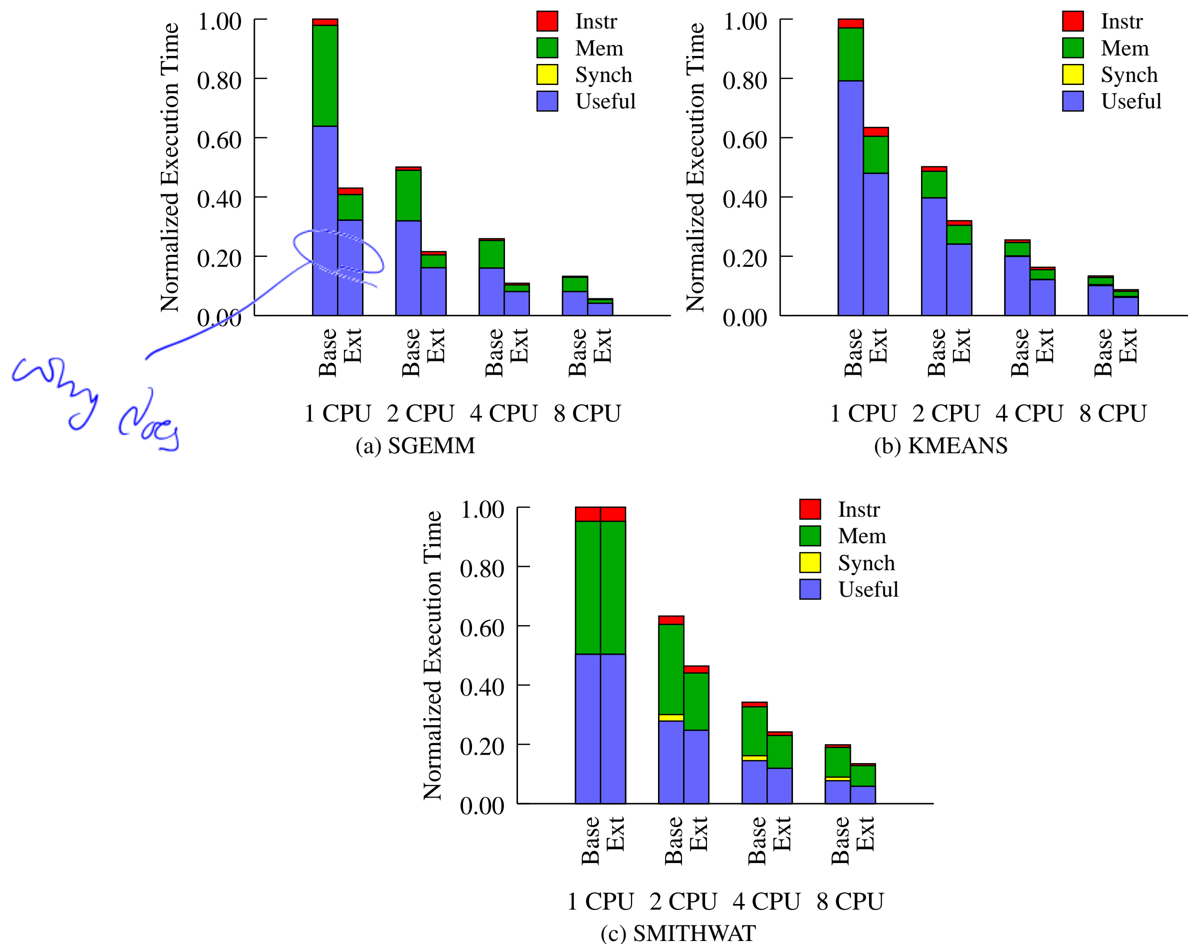


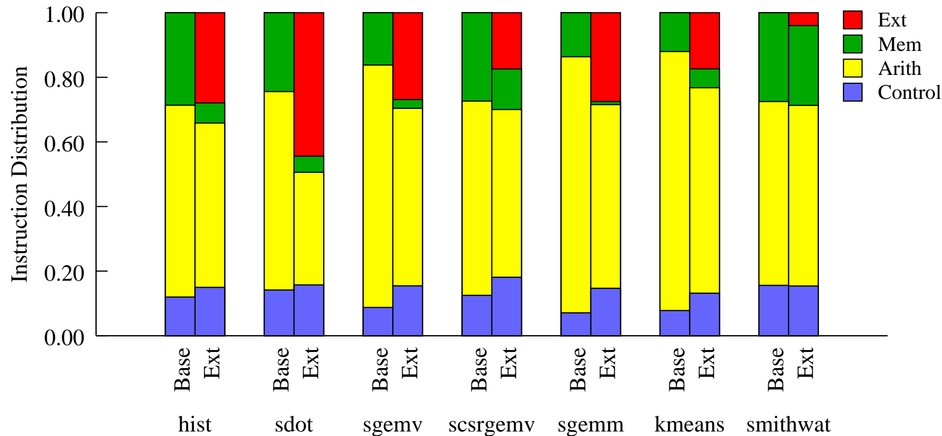
Figure 5.8: Execution time of benchmarks on both base and extended CMPs. Time normalized to the base CMP with a single processor core. Continued from Figure 5.7.

these instructions incur that same costs as all others, they are often the same instruction (a load or store to a fixed address) and can be more easily optimized away as will be shown in following sections.

For some benchmarks the reduction in instructions fetched is complemented by reductions in the number of accesses to the L1 D-cache (another significant energy consumer). In **SMITHWAT**, the direct inter-core communication provided by the Ring network facilitates an alternate formulation of the dynamic programming algorithm that reduces the number of data memory accesses by 24%. The reduction in data memory accesses further improves energy efficiency and complements the reduction in instructions executed and the improvement in performance.

hist	sdot	sgemv	scsrgemv	sgemm	kmeans	smithwat
0.86	0.89	0.56	0.64	0.47	0.39	0.73

(a) Instructions executed on 8 core extended CMP normalized to base CMP



(b) Distribution of instruction types executed on 8 core base and extended CMPs

Figure 5.9: Amount and distribution of types of instructions executed on 8 core extended and base CMPs. `Ext` indicates extension instructions, while `Mem`, `Arith`, `Control` are memory accesses, arithmetic instructions and control flow instructions respectively.

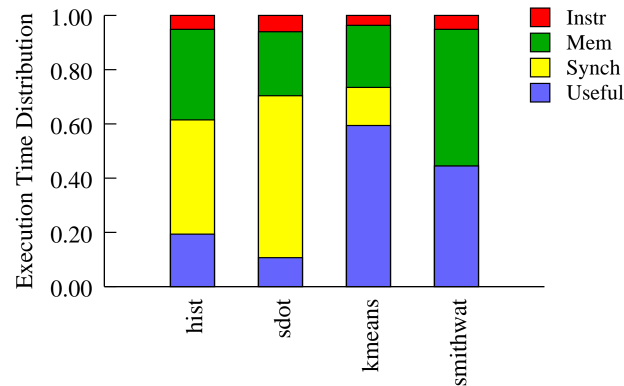
The experimental benchmarks were chosen to capture a range of different behaviors. Like many informatics applications that are characterized by large structured memory accesses and significant thread interaction. Applications with few structured accesses and straightforward parallelism will not benefit as much or as consistently from the example extended architecture. In general, in cases where the extensions might be of uncertain value, the backward compatibility of the extended CMP ensures that user can continue to use existing implementations, and will not be forced to accept performance less than is possible with the base CMP.

5.4.3 Performance Implications of Backward/Inter Compatibility

Section 5.3.3 motivated the use of coarse-grain variant selection as opposed to emulation for backward or inter-compatibility. When the extension functionality is significantly different than that provided by the base architecture, such as for the Ring and TIU extensions, emulated kernels can be significantly slower than those purposely implemented for the base CMP. In order to quantify these effects, alternate versions of four benchmarks, **HIST**, **SDOT**, **KMEANS** and **SMITHWAT**, were

	hist	sdot	kmeans	smithwat
Base	0.52	0.42	0.50	0.63
Ext	0.43	0.31	0.32	0.46
Emul	3.73	5.42	0.73	1.13

(a) Relative execution time



(b) Execution time distribution

Figure 5.10: Relative execution time normalized to one base core and time distribution for emulated extension features executing on two cores

implemented with libraries that emulated the extension's features. Figure 5.10 shows the relative execution time and the execution time distribution for emulated versions of four benchmarks running on two processor cores. In Figure 5.10a, execution time is normalized to the base CMP-only code (*i.e.* non-emulated) executing on one core.

When targeting the extended CMP, **HIST**, **SDOT** and **KMEANS** make extensive use of the TIU and Fetch extensions. The differences between the purpose-built implementation and emulated implementation are minimal for the Fetch extension; the structured vector accesses are simply converted to the corresponding indexed array access. The TIU, however, allows very low cost global accumulation that has no counterpart in the base CMP, and must instead be implemented with expensive locking operations. When there is extensive sharing of a few global locations, such as in **HIST** and **SDOT**, there are significant performance penalties for emulation, with the benchmark actually executing many times slower on the two cores, than on a single core. As Figure 5.10b shows, the **HIST** and **SDOT** kernels spend a significant amount of time waiting for access to the shared memory locations.

The low cost thread interaction provided by the TIU enables the program to be effectively parallelized at a much finer grain than is appropriate for the base CMP. For the base CMP, the programmer or compiler must collapse the data-level parallelism into very coarse grain threads, and generate thread-private temporaries to minimize thread interaction. For the simple atomic add operations required by these kernels it is possible for the compiler to be able to generate the necessary coarse-grain threads and local accumulators; however, for complex thread interactions, sophisticated algorithmic transformations, beyond those the compiler is capable of, might be required. **SMITHWAT** is an example of such a kernel.

SMITHWAT computes the entries in a two-dimensional score array using a dynamic programming algorithm, in which array entry (i, j) is dependent on entries $(i-1, j-1)$, $(i-1, j)$ and $(i, j-1)$. When the Ring extension is available, adjacent rows are mapped to adjacent processors, with the dependent array entries passed via the Ring FIFO. When this algorithm is emulated, however, using software FIFOs, there is significant overhead resulting in a slowdown on multiple cores. The base CMP-targeted implementation uses a coarse-grain blocked parallel decomposition that maximizes reuse and minimizes sharing. The two approaches are very different, and it would be difficult for the compiler to automatically transform the Ring-based version to the blocked version. In this case, coarse-grain code selection is required to ensure good performance on both base and extended CMPs.

5.4.4 Invasive Microarchitectural Changes

The modular microarchitecture as described does not require any changes to the processor datapath or instruction decoder; however, as suggested in Section 5.4.2, there are additional optimizations possible when changes are permitted. When used in direct access mode, accesses to the extensions are highly stereotyped. In the case of reads from the Fetch interface queues, each iteration of a loop will read from the same fixed address. Instead of issuing a separate load for each read operation, alternate register semantics could be defined, where the read from a register automatically triggers an extension-targeted memory access. Register reads and writes would become configurable, and would either directly access the traditional register file, or generate the appropriate static memory request.

When there is no reuse within the register file of the values retrieved from the extension, as is often the case for dense matrix computations and other kernels, using the register file as an indirect addressing structure can improve performance and energy efficiency. Figure 5.11 shows the normalized execution time of register mapped extension accesses (performance is relative to

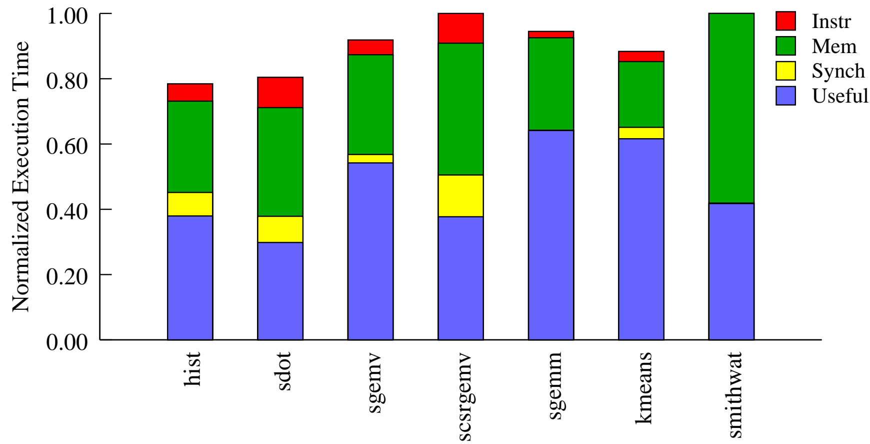


Figure 5.11: Relative execution time for direct register mapping vs. memory mapped for an 8 core extended CMP

memory mapped extensions). As expected, those benchmarks that execute more extension targeted instructions generally show a larger performance improvement. Eliminating these instructions will further reduce the total number of instructions executed, improving energy efficiency.

Although using register-mapping requires changes to the core datapath, backward compatibility for legacy software is preserved. Further in contrast to application-specific instruction set processors (ASIPs) [24] the changes to the ISA and datapath are not specific to any one extension or application domain. The register mapping logic is implemented once and can be used by many different extensions, including those to be developed in the future. In this way the modular microarchitecture can offer direct register-level access to extensions similar to that in ASIPs without needing to customize the processor datapath or instruction decode for each extension (as is required in non-hardware reconfigurable ASIPs). More such optimizations are possible and are the subject of future work.

Chapter 6

Conclusion

The combination of exponentially growing datasets and more complex data analysis, mining and synthesis algorithms are driving an exponential growth in computational requirements. Datasets are now growing faster than the performance of general-purpose processors (GPPs). Heterogeneous multicore computers can provide the scalable performance required by demanding informatics applications. Relative to GPPs, heterogeneous systems can deliver orders-of-magnitude better computational performance and dollar, energy and volume efficiency. Programming such systems is a keen challenge, however, and new tools are needed to make these hardware platforms accessible to informaticians, and other domain experts.

This thesis proposed Merge, a general-purpose programming model for heterogeneous multicore computers. Merge provides a pragmatic, library metaprogramming-based approach to building applications for heterogeneous systems. Specialized domain-specific languages or accelerator specific assembly are encapsulated in C/C++ functions to provide a uniform interface and inclusive abstraction for computations of any complexity. Different implementations of a function are bundled together, creating a layer of indirection between the caller and the implementation that facilitates the automatic mapping between the application and implementation.

Merge differs from other library metaprogramming tools by performing dynamic specialization (of generic function calls to specific implementations) using descriptive library-based annotations. Specialization is effected by dispatch meta-wrappers generated by the compiler using the input and machine configuration annotations supplied with each function implementation. Implicit in the implementation of these meta-wrappers is the objective function that controls the mapping of the function call to an implementation. The meta-wrappers hide the complexity of mapping applications to complex heterogeneous systems from the programmer, making these applications more extensible

and easier to maintain than approaches that use static, prescriptive, application-based specialization.

The tradeoff is that the user is dependent on the compiler (and runtime) to generate a "good" objective function. However, the performance results in Chapter 4 show that programmers can effectively exploit heterogeneous systems even using very simple inferred objective functions. Further exploring the performance implications of Merge's dispatch system for new and different applications, targeting new and different hardware platforms, is an important area of ongoing research.

Looking forward I argue for an approach to programming for heterogeneous systems that is modeled on the implementation of the map-reduce language in this thesis. Such an approach would combine direct usage of the Merge function bundles for low-effort, low-complexity access to heterogeneous hardware, with parallel programming frameworks that leverage Merge's encapsulation and metaprogramming capabilities to provide high-level domain-specific languages (DSLs), sophisticated schedulers and other tools that make heterogeneous systems more accessible.

High-level parallel frameworks are most effective for the broad middle ground between GPPs and bleeding-edge accelerators. GPPs are readily targeted using conventional programming tools, while new accelerators invariably lack sophisticated compiler support and must be programmed using accelerator-specific assembly or other similarly low-level tools. Merge enables implementations targeting both systems to coexist; applications can exploit the newest and most powerful computing resources without compromising performance on legacy architectures. By also incorporating DSL-based function implementations, programmers can leverage steadily advancing compiler technology to improve productivity and performance for established heterogeneous systems.

Support for multiple implementations for the same computation relaxes the pressure on the programmer and the compiler designer to produce the one "best" optimized implementation from an encapsulated DSL. The product of the DSL compiler, a new function-intrinsic, will just be one of possibly several different implementations for a computation. The DSL compiler does not need to generate the one best implementation for all scenarios. Instead it can focus on generating an optimized implementation for a particular input or machine configuration. Tasks that would be common to many such tools, such as performance ranking different function variants, would be provided as part of the Merge infrastructure. With a focused mission and powerful supporting infrastructure, high-level parallel frameworks would be simpler and easier to build, accelerating the development of sophisticated compiler support for new and evolving hardware.

Bibliography

[1] S. Agrawal, W. Thies, and S. Amarasinghe. Optimizing stream programs using linear state space analysis. In *Proc. of CASES*, pages 126–136, 2005.

[2] E. Allen, D. Chase, J. Hallet, V. Luchangco, J-W Maessen, S. Ryu, G. L. Steele, and S. Tobin-Hochstadt. The Fortress language specification version 1.0beta. Technical report, Sun Microsystems, 2007.

[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, University of California, Berkeley, 2006.

[4] T. Austin, E. Larson, and D. Ernst. Simplescalar: an infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, 2002.

[5] J. Balfour, W. J. Dally, D. Black-Schaffer, V. Parikh, and J. Park. An energy-efficieny processor architecture for embedded systems. *Computer Architecture Letters*, 7(1):29–32, 2008.

[6] C. Barret and S. Berezin. CVC lite: A new implementation of cooperating validity checker. In *Proc. of Conf. on Computer Aided Verification*, pages 515–518, 2004.

[7] L. A. Barroso and U. Hölzle. The case of energy-proportional computing. *IEEE Computer*, 40(12):33–37, 2007.

[8] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proc. of PPoPP*, pages 207–216, 1995.

[9] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[10] F. Campi, R. Canagallo, and R. Guerrieri. IP-reusable 32-bit VLIW RISC core. In *Proc. of European SSCC*, pages 456–459, 2001.

[11] S. Chiba. A metaobject protocol for C++. In *Proc. of OOPSLA*, pages 285–299, 1995.

[12] S. Choi, N. Kohout, S. Pamnani, D. Kim, and D. Yeung. A general framework for prefetch scheduling in linked data structures and its application to multi-chain prefetching. *ACM Trans. on Computer Systems*, 22(2):214–280, 2004.

[13] CodeSourcery. Sourcery VSIPL++. http://www.codesourcery.com/vsiplplusplus.

[14] Intel Corporation. Intel's next generation integrated graphics architecture – intel graphics and media accelerator X3000. Technical report, Intel Corporation, 2006.

[15] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. of OSDI*, pages 137–149, 2004.

[16] G. Diamos and S. Yalamanchili. Harmony: An execution model and runtime for heterogeneous many core systems. In *Proc. of HPDC*, pages 197–200, 2008.

[17] P. Dubey. Recognition, mining and synthesis moves computers to the era of tera. *Technology@Intel Magazine*, 2005.

[18] M. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *European Conf. on Object-Oriented Programming*, pages 186–211, 1998.

[19] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood. Lx: A technology platform for customizable vliw embedded processing. In *Proc. of ISCA*, pages 203–213, 2000.

[20] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequioa: Programming the memory hierarchy. In *Proc. of ACM/IEEE Conf. on Supercomputing*, page 83, 2006.

[21] W. E. Jr. Ferguson. Selecting math coprocessors. *IEEE Spectrum*, 28(7):38–41, 1991.

[22] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proc. of the IEEE*, 93(2):216–231, 2005.

[23] P. Gepner, D. L. Fraser, and M. F. Kowalik. Second generation quad-core intel xeon processors bring 45nm technology and a new level of performance to HPC application. In *Proc. of ICCS*, pages 417–426, 2008.

[24] R. E. Gonzales. Xtensa: A configurable and extensible processor. *IEEE Micro*, 20(2):60–70, 2000.

[25] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *Proc. of ASPLOS*, pages 291–303, 2002.

[26] S. Z. Guyer and C. Lin. Annotation language for optimizing software libraries. In *Proc. of Conf. on Domain Specific Languages*, pages 39–52, 1999.

[27] S. Z. Guyer and C. Lin. Optimizing high performance software libraries. In *Proc. of 13th International Workshop on Languages and Compiler for Parallel Computing*, 2000.

[28] R. Hankins, G. Chinya, J. D. Collins, P. Wang, R. Rakvic, H. Wang, and J. Shen. Multiple instruction stream processor. In *Proc. of ISCA*, pages 114–127, 2006.

[29] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *Proc. of DATE*, pages 642–649, 2001.

[30] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*, volume 4. Morgan Kaufmann, 2007.

[31] H. P. Hofstee. Power efficient processor architecture and the cell processor. In *Proc. of HPCA*, pages 258–262, 2005.

[32] IBM. *Basic Linear Algebra Subprograms Library Programmer's Guide and API Reference*, sc33-8426-01 edition, 2008.

[33] Intel. Intel C++ compiler. http://www3.intel.com/cd/software/products/asmo-na/eng/compilers/284132.htm.

[34] Intel. Intel threading building blocks. http://www3.intel.com/cd/software/products/asmo-na/eng/294797.htm.

[35] U. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. Owens. Programmable stream processors. *IEEE Computer*, 36(8):54–62, 2003.

[36] K. Kennedy, B. Broom, A. Chauhan, R. Fowler, J. Garvin, C. Koelbel, C. McCosh, and J. Mellor-Crummey. Telescoping languages: A system for automatic generation of domain languages. *Proc. of the IEEE*, 93:378–408, 2005.

[37] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[38] R. Krashinksy, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic. The vector-thread architecture. In *Proc. of ISCA*, 2004.

[39] V. Kuncak, P. Lam, and M. Rinard. Role analysis. *ACM SIGPLAN Notices*, 37:17–32, 2002.

[40] S. Leibson and J. Kim. Configurable processors: A new era in chip design. *IEEE Computer*, 38(7):51–59, 2005.

[41] P. Lyman and H. R. Varian. How much information. http://www.sims.berkeley.edu/how-much-info-2003, 2003.

[42] W. Mark, R. Glanville, K. Akeley, and M. Kilgard. Cg: A system for programming graphics hardware in a C-like language. *ACM Trans. Graph.*, 22(3):896–907, 2003.

[43] M. McCool and S. Toit. *Metaprogramming GPUs with Sh*. A K Peters, 2004.

[44] M. McCool, K. Wadleigh, B. Henderson, and H. Y. Lin. Performance evaluation of GPUs using the RapidMind development platform. In *Proc. of ACM/IEEE Conf. on Supercomputing*, page 81, 2006.

[45] T. Millstein. Practical predicate dispatch. In *Proc. of OOPSLA*, pages 345–264, 2004.

[46] A. Nayak, M. Haldar, A. Kanhere, P. Joisha, N. Shenoy, A. Choudhary, and P. Banerjee. A library based compiler to execute MATLAB programs on a heterogeneous platform. In *Proc. of Conf. on Parallel and Distributed Computing Systems*, 2000.

[47] J. Nickolls and I. Buck. NVIDIA CUDA software and GPU computing platform. In *Microprocessor Forum*, 2007.

[48] NVIDIA. *CUDA CUBLAS Library*, 2.0 edition, March 2008.

[49] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*, 2.0 edition, 2008.

[50] S. Oberman, G. Favor, and F. Weber. AMD 3DNow! technology: Architecture and implementations. *IEEE Micro*, 19(2):37–48, 1999.

[51] M. Oka and M. Suzuoki. Designing and programming the emotion engine. *IEEE Micro*, 19(6):20–28, 1999.

[52] M. Oskin. Sim3. http://www.cs.washington.edu/homes/oskin/tools.html.

[53] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell. A survery of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[54] Peakstream. The PeakStream platform: High productivity software development for multi-core processors. Technical report, PeakStream Inc., 2006.

[55] C. Price. *MIPS IV Instruction Set*. MIPS Technologies, Inc., Mountain View, CA, 3.2 edition, September 1995.

[56] S. K. Raman, V. Pentkovski, and J. Keshava. Implementing streaming simd extensions on the pentium III processor. *IEEE Micro*, 20(4):47–57, 2000.

[57] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proc. of HPCA*, pages 13–24, 2007.

[58] M. Segal and M. Peercy. A performance-oriented data parallel virtual machines for GPUs. Technical report, ATI Technologies, 2006.

[59] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[60] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: Using data parallelism to program GPUs for general-purpose uses. In *Proc. of ASPLOS*, pages 325–335, 2006.

[61] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. CACTI 4.0. Technical Report HPL-2006-06, HP, 2006.

[62] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. In *Proc. of IEEE Micro*, pages 25–35, 2002.

[63] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Proc. of Conf. on Compiler Construction*, pages 49–84, 2002.

[64] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G-Y Lueh, and H. Wang. EXOCHI: Architecture and programming environment for a heterogeneous multi-core multithreaded system. In *Proc. of PLDI*, pages 156–166, 2007.

[65] P. H. Wang, J. D. Collins, G. N. Chinya, B. Lint, A. Mallick, K. Yamada, and H. Wang. Sequencer virtualization. In *Proc. of ICS*, pages 148–157, 2007.

[66] O. Wechsler. Inside intel core micoarchitecture: Setting new standards for energy-efficient performance. Technical report, Intel, 2006.

[67] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Ccomputing*, 27(1-2):3–35, 2001.

[68] L. Wu, C. Weaver, and T. Austin. Cryptomaniac: A fast flexible architecture for secure communication. In *Proc. of ISCA*, pages 110–119, 2001.

[69] D. Zhang, Z. Li, H. Song, and L. Liu. *Embedded Computer Systems: Architectures, Modeling and Simulation*, chapter A Programming Model for an Embedded Media Processing Architecture, pages 251–261. Springer, 2005.

[70] Zachary S Zumsteg, Caleb Kemere, Stephen O'Driscoll, Gopal Santhanam, Rizwan E Ahmed, Krishna V Shenoy, and Teresa H Meng. Power feasibility of implantable digital spike sorting circuits for neural prosthetic systems. *IEEE Trans Neural Syst Rehabil Eng*, 13(3):272–279, 2005.