16th International Conference on Parallel Architecture and Compilation Techniques

CIGAR: Application Partitioning for a CPU/Coprocessor Architecture

John H. Kelm*, Isaac Gelado†, Mark J. Murphy*, Nacho Navarro†, Steve Lumetta*, Wen-mei Hwu*

*Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
{jkelm2, mjmrphy1, steve, hwu}@crhc.uiuc.edu

†Computer Architecture Department
Universitat Politecnica de Catalunya (UPC)
{igelado, nacho}@ac.upc.edu

Abstract

We present CIGAR, a methodology and development platform that facilitates the use of data-parallel coprocessors. With CIGAR, application developers use profiling tools to identify parts of the application for data-parallel execution, determine the application data structures to be hosted by the coprocessor, prototype coprocessor execution of these parts, and debug correctness of partitioned execution of the application using emulation.

The CIGAR methodology is complemented by a CPU/FPGA prototyping platform that runs a fully functional version of the Linux operating system and associated development tools and libraries. To guide the development of our work and to evaluate its utility, we have instrumented SPECint2006 applications to utilize coprocessors emulated by softcore processors embedded in our prototyping platform. Examples of how a developer would use CIGAR to partition an application for a heterogeneous CPU/coprocessor environment are demonstrated.

1. Introduction

This paper presents the CUBA Infrastructure for Guided Application Remapping (CIGAR) application partitioning methodology for general-purpose processors that incorporate data-parallel coprocessors. CIGAR is demonstrated using a prototype of the Champaign-Urbana-Barcelona Architecture (CUBA) CPU/coprocessor architecture, which we are developing. The key feature of CUBA is that coprocessor-accessible data structures are *hosted* by memory local to the coprocessor, but are still accessible by the general-purpose processor. Together, the techniques we describe enable a *software* developer to take a piece of software and map it to a heterogeneous multicore system. Moreover, CIGAR lays the groundwork necessary to enable a methodical and automated approach to application partitioning for CPU/coprocessor systems with hosted data structures.

The first contribution of this work is the CIGAR methodology which leverages both developer knowledge of an application and dynamic application profiling techniques to partition data applications for CPU/coprocessor systems. The second contribution is an emulation platform for rapid prototyping of CPU/coprocessor applications being partitioned with CIGAR. The platform provides enhanced visibility, control, and speed to the software developer, with the added capability of emulating CPU/coprocessor systems that are still under development.

1.1 Background

A variety of approaches have been proposed to extract greater performance from the increasing number of transistors available to microprocessor designers. One example is homogeneous multicore designs integrating multiple, identical cores on a single die [9, 18]. These designs are adept at exploiting thread-level parallelism, but are not the most power-efficient computational substrate for data-parallel codes. Several efforts have explored heterogeneous systems as a means to gain better performance with greater power and area efficiency. The Cell processor integrates groups of smaller, in-order vector units with a high-performance superscalar core [3]. To avoid the need for a heterogeneous programming model, other work has investigated the use cores of varying capabilities from the same instruction set architecture (ISA) on a die [11]. Furthermore, heterogeneous multicore processors are shown to provide favorable power/performance benefits to a wide array of applications [10], but may still be limiting for irregular, data-parallel codes.

An alternative approach is to use coprocessors that are specifically designed to exploit data parallelism with a high level of power and area efficiency. Specifications are already underway to incorporate fine-grained examples of such coprocessors into commodity microprocessors [14]. The goal of such systems is to complement high-performance processor cores by incorporating domain-specific functionality that is implemented as fast, closely-coupled logic (e.g., [15]). For a study of data parallelism and its effect on microarchitecture, see [16]. The end result is a heterogeneous mix of general-purpose and coprocessor cores that can exploit the various forms of parallelism present in applications [1].

We define a coprocessor as a programmable set of functional units, possibly with its own instruction memory, that is under the control of a general-purpose processor. The functional units of the coprocessor are chosen to provide high performance for a certain class of applications, but may be unable to do arbitrary arithmetic operations. We further restrict the focus of coprocessors in this work to data-parallel coprocessors that are intended to accelerate the computation of applications with phases in which there are a large number of arithmetic operations. These operations could be executed in parallel, but are unnecessarily serialized due to the restricted number of functional units available in a general-purpose

1089-795X/07 $25.00 © 2007 IEEE
DOI 10.1109/PACT.2007.13

IEEE computer society

processor (CPU). Furthermore, in our model coprocessors have their own physical address space, called coprocessor local memory (CLM), that can be mapped to both the CPU and the coprocessor.

The design of interconnect between the CPU and the coprocessors is critical to the performance achievable by the system. Commercial examples of coprocessor interconnects include system buses (e.g., HyperTransport and PCI Express) and instructions provided by the ISA (e.g., MIPS Coprocessor Interface). System bus interfaces provide high-bandwidth, high-latency connections between system memory, CPU, and coprocessors without imposing upon the ISA. On the other hand, commercially-available ISA extensions provide a low-latency access mechanism for coprocessors, but with register granularity that requires entangling coprocessor interfaces with the processor pipeline. Both models provide *pass-by-value* semantics whereby the data is explicitly delivered to the coprocessor and the CPU does not keep a reference to the data. The CIGAR approach introduced in this paper is based on a coprocessor architecture that avoids modifying the ISA of the general-purpose CPU and utilizes a *pass-by-reference* model where persistent data is shared between the CPU and coprocessor.

Several research platforms have proposed the integration of application-specific coprocessors with general-purpose processors. The GARP project [4] proposes a reconfigurable coprocessor connected to a MIPS general-purpose processor for bit-level computations as an extension of the ISA. The OneChip [23] prototype is a similar platform where coprocessors implement fine-grained controllers and accelerators while accounting for memory consistency between the CPU and coprocessors [6]. The MorphoSys platform [17] implements coprocessors with word-level computation accessed via SIMD-like instructions. The main difference between the CIGAR model and these previous coprocessor interconnect designs is the concept of data structure hosting in coprocessors as the mechanism for CPU/coprocessor data transfers.

A commercial example of a data-parallel coprocessor and development environment is the NVIDIA G80 and its corresponding CUDA [19] software development environment. CUDA is an environment for developing software that will run on the coprocessor; However, it currently does not provide a means to determine an appropriate partitioning of applications across the CPU and coprocessor as CIGAR achieves. The key difference between the G80 model and CUBA is that the G80 provides only a pass-by-value model for the CPU to access its local memory, while CUBA provides both pass-by-value and, as we use extensively in this study, pass-by-reference semantics.

In order to effectively use the data hosting feature, the application and its data must be properly partitioned between the CPU and coprocessors. Application partitioning has been widely studied in the field of design automation. The FLAT tool set [20] uses source code profiling and simulation to identify the compute-intensive loops of applications. In the field of application-specific instruction set processors (ASIP), the μP tool set [7] uses fine-grained assembly-level profiling and simulation to identify instruction extensions for general-purpose cores. The development environment for Stretch [2] is an example of using simulation and native execution to rapidly prototype and debug CPU/coprocessor designs. While the Stretch tool flow allows for direct performance measurements, it requires possibly time-consuming synthesis and place-and-route steps to be performed before evaluating a design. Their model does not use a prototyping mechanism to reduce the time spent debugging large-scale coprocessor designs as we aim to do via emulation.

1.2 Motivation

For heterogeneous CPU/coprocessor systems to become prevalent, methodologies such as CIGAR must be developed that allow software designers, with as little added effort as possible, to take common applications and partition them across CPUs and coprocessors. Such tools should not deviate from accepted development and debugging practices while also allowing the developed applications to remain portable. We explore partitioning techniques that require programmer intervention, but defer time consuming tasks as much as possible to converge on a correct design more rapidly.

The CUBA architecture and the CIGAR methodology are designed to help software developers achieve three important objectives when utilizing data-parallel coprocessors. First, data sharing should consume the least possible amount of interconnect bandwidth and incur shortest possible latencies. Second, the smallest possible number of changes should be made to the programming environment and processor architecture so that software can be easily ported between systems with and without coprocessors. Third, the software developed using the tools should be easy to debug. The latter two objectives are achieved partly by providing an emulation platform that eases debugging the communication between the part of the application running on the general-purpose core and that executed by the coprocessor. It should be noted that CIGAR is not meant to evaluate the *performance* of a partitioned design, but instead has the orthogonal goal of developing a *correct* design.

The CIGAR flow starts with a software application that runs on the general-purpose processor. The application developer then uses CIGAR to *identify* the parts of the application that are amenable to coprocessor execution and *select* the data structures that are accessed frequently by those software components. The developer can then use our emulation platform to test, debug, and verify both software and hardware-instrumented versions of the application therby doing code *generation*. Our emulation platform allows for the flow to be completed, *after* the hosted data mappings and synchronization mechanisms are in place and debugged.

Case studies are presented to demonstrate the use of CIGAR and CUBA to enable the efficient use of data-parallel coprocessors. We choose three SPEC2006 integer benchmarks to map to our emulation platform using our analysis and profiling infrastructure: one to guide the development of CIGAR and two others to verify its capabilities. We show that a software designer can take a software application and quickly prototype designs that incorporate possibly non-existent coprocessors with high visibility for debugging purposes.

1.3 Contributions

The five contributions of this work are as follows:

1. An architecture that enables a novel method of coprocessor memory access that brings data closer to the coprocessor resource accessing it.
2. A profiling methodology that assists software developers in mapping software applications and data objects into CPU/coprocessor architectures.
3. A rapid prototyping environment that allows for the debug and test of CPU/coprocessor designs.
4. An emulation technique that uses softcore processors to avoid the time consuming process of hand implementation, or high-level synthesis, of coprocessors during partitioning.
5. A design flow that starts with a purely software application and results in that application debugged, tested, and partitioned to run on a CPU/coprocessor architecture.

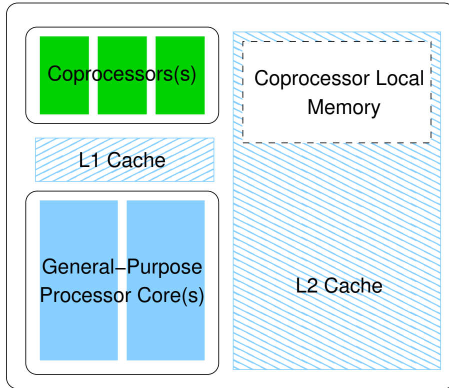

Figure 1. System Model Overview

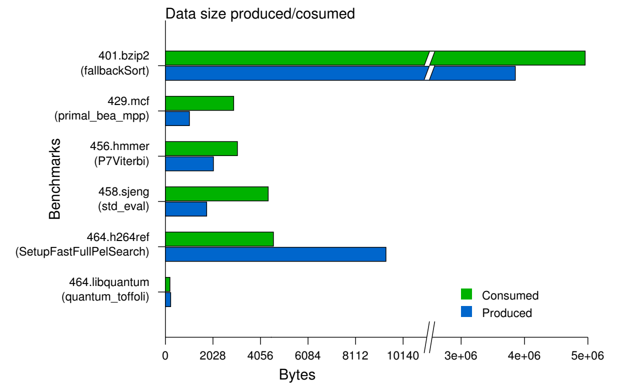

Figure 2. Data Accessed by Accelerator Candidates

The remainder of the paper is organized as follows: Section 2 gives an overview of CUBA. Section 3 presents CIGAR. Section 4 presents a set of case studies using CIGAR and our emulation platform. Section 5 concludes the paper.

2. CPU/Coprocessor Architecture

Software applications mapped into data-parallel coprocessors can incur high startup overhead when control is transferred from the CPU to the coprocessor. Furthermore, traditional partitioning methods require changes be made to the code that isolate the data transferred to the coprocessor, a technique known as data marshaling. *Hosting* data structures in memories local to coprocessors, as is done in CUBA, is a technique that can address both issues. Moving data to memories close to the coprocessor as that data is produced can reduce overhead of explicit copies. Marshaling overhead is reduced by directly mapping application data structures to coprocessor memories. In this section we give an overview of features present in such an architecture including: coprocessor memories that host application data structures; data flow transfers; synchronization; exception models; concurrent CPU and coprocessor execution; and memory coherence mechanisms.

2.1 Data structure hosting

We define data structure hosting as the placement of data into a localized memory, such as a coprocessor-local scratchpad memory or specially partitioned/tagged region of the processor data cache, *without keeping a coherent copy in a backing store, such as system memory*. When an application data structure is hosted by a coprocessor local memory, it is temporarily removed from the memory coherence domain of the CPU(s). A block diagram of our model is shown in Figure 1. The hosted data is assumed to be accessed by the coprocessor with latency comparable to an L1 cache hit and accessed by the general-purpose processor, via the cache hierarchy, with a latency comparable to an L2 cache hit. The mechanism is intended to provide better data locality for coprocessor function units, but still allow the CPU to access the data in an efficient way.

What follows is a description of the coprocessor calling mechanism. At the start of a call, any cached data that is hosted by the coprocessor local memory (CLM) is explicitly flushed thus ensuring coherence for data passed both by-value and by-reference. The CPU transfers control to the coprocessor by writing the opcode specifying which coprocessor function to perform to a memory-mapped register in the CLM address space. As soon as the write to the register completes, the coprocessor begins execution. The CPU can then poll when it needs the result, waiting for a status register to be set by the coprocessor, signifying that the coprocessor has completed execution and the results are available in the CLM. We do not investigate interrupt-driven mechanisms in this study. Values are returned either implicitly by-reference, as the hosted data structures are updated directly by the coprocessor, or explicitly by-value for non-hosted data as part of the CLM using a calling convention we define.

In a one-to-one coprocessor+CPU setup, maintaining global coherence can be avoided for data placed in the CLM. However, for coprocessors with multiple input sources in a multicore processor system, it will be necessary to have a mechanism, such as a modified write-update policy, applied to cache lines resident in CLM. In our prototype, coherence is maintained by explicitly flushing cache lines from the CPU cache prior to use by the coprocessors. An implication of explicit cache coherence prior to coprocessor computation is the possibility for simplified coprocessor logic due to deterministic access times to the CLM since all data is available prior to use. Other coherence mechanisms are left to future work.

2.2 Application interface

The application programming interface (API) that the programmer sees is that of a calling convention whereby parameters not hosted by the CLM are passed by-value and those hosted by the coprocessor are passed by-reference as a pointer into the CLM. Parameters passed by-value are written into locations defined during partitioning that are inside the CLM and are known by both the CPU and the coprocessor. When the application makes the call to the coprocessor, the programmer must ensure that:

1. the data being passed to the coprocessor is available, that is, the computation that produces the value precedes the coprocessor call in program order,
2. any pointers in any of the parameters are within the address space of the CLM, which is a form of marshaling that is handled via simple macros in the code, and
3. none of the hosted data structures have elements that are treated as pointers and dereferenced by the coprocessor that are modified by any other processing element in the system.

One last programming consideration is that the CLM is a limited resource and must be virtualized to allow for wide-ranging

data structure sizes and implementation flexibility. However, for many applications the complexity of virtualization can be elided by remapping. Figure 2 shows the amount of data produced and consumed by the most compute-intensive subroutines of selected SPEC2006 integer benchmarks. The data accessed by each call correlates with the data that would be passed by-value and by reference in our model. If these functions were implemented as coprocessors, for many of our benchmarks, the area required to host the relevant data structures is less than the memory dedicated to the L2 cache of a single core of a multicore processor today.

2.3 Execution modes

In our model, we define two possible modes of concurrent execution during the *start-to-poll interval*, or the period between when the CPU starts the coprocessor computation and when the CPU regains control upon coprocessor completion. The modes include: 1) Independent Execution Mode, and 2)Exception Handling Mode.

In independent execution mode, during start-to-poll interval for the application the CPU is allowed to execute while the coprocessor computation completes. The goal of the independent scheduling mode is to avoid wasted cycles polling. The method provides as much overlap as there are independent instructions capable of being placed in the start-to-poll interval of the schedule.

Exception handling mode allows for simplified hardware coprocessors by removing difficult or infrequent control paths inside the coprocessor (e.g., exceptions such as divide by zero) and having the CPU speculatively execute every coprocessor task during the start-to-poll interval. We assume that the ability to guard against undesirable control paths in the kernel of execution is more efficient than doing the actual computation we avoid implementing. The CPU can continue to check the completion of the coprocessor periodically and, if the CPU reaches the end of the computation ahead of the coprocessor, the CPU squashes the coprocessor assuming an exception has occurred and commits its own results. One disadvantage of this mode, however, is that two versions of the data must be kept should invalid results from the coprocessor need to be flushed.

An example illustrating where exception handling mode may be useful is in the `quantum_sigma_x()` function from 462.libquantum. In this example there is a control path that is never taken for certain inputs. We leverage the fact that a check at the start of the function always returns true to reduce the complexity of the coprocessor logic. If the library were implemented using coprocessors, then the avoided control path can be executed on the CPU and the coprocessors actions can be squashed when the check does return false. The model requires copies of modified data to be maintained, which could be implemented by keeping a separate copy in the CPU cache similar to hardware transactional memory models [5, 8], but due to limited space we do not explored that posibility.

2.4 Summary

An overview of an architecture has been presented for CPU/coprocessor systems that host data structures, accessed by the coprocessor, in coprocessor local memories closely coupled to the processor core. The required semantics related to transfer of ownership have been presented for both synchronization and data flow between CPUs and coprocessors. We provide possible modes of execution that increase concurrency and enable simplified coprocessors to be built.

Benchmark	Total Instrs.	ALU Instrs.	Loop-body DLP	Cross-iteration Mechanism
libquantum	8	3	3	Loop Unrolling
hmmer	84	22	40	Loop Skewing
h264ref	223	62	90	Streaming

Table 1. Data-level parallelism present in loop bodies and mechanisms for exploiting the cross-iteration parallelism

3. Design Flow

We present the CIGAR design flow, which aids developers in partitioning applications on to CPU/coprocessor architectures. We then describe a prototype emulation platform that supports rapid prototyping and debugging of alternative partitioning strategies. The prototyping platform supports a novel emulation technique that allows us to perform substantial testing and validation of partitioning prior to the availability of coprocessor hardware implementations. We use our prototyping platform to emulate the CUBA architecture, which is an example of our CPU/coprocessor model.

3.1 Analysis and profiling tools

CIGAR is a methodology that enables a software developer to identify the subroutines and data structures amenable to coprocessor implementation and data structure hosting, respectively. The CIGAR approach arose from our experiences partitioning a SPEC application, 462.libquantum. To demonstrate the developed methodology, we apply CIGAR to two other benchmarks.

A high-level diagram of the design process appears in Figure 3. The application (upper left corner) is first fed into a suite of profiling and analysis tools. The developer uses the analysis as a guide to selecting both functions and data objects, which are then executed and hosted by the coprocessor, respectively. The developer's choices are then used to partition the program, producing three results: stub code for controlling the interaction between the general-purpose processor and the coprocessor; a data structure mapping that controls placement of objects into the CLM as well as the associated allocation management; and an implementation of the coprocessor functions suitable for emulation by a softcore processor. The tool flow (the block marked "Profiling/Analysis Tools" in Figure 3) works as follows: First, subroutine candidates are found by profiling the application and determining the data parallelism present. Data to be hosted by CLM is discovered by correlating the load/store intensity of dynamically allocated data structures with the execution periods of high computation time subroutines. We provide a means to vary the resolution of our tools to enable quicker analysis while searching for interesting regions of execution and slower, in-depth analysis of those regions found with coarser resolution. The visualizations produced by CIGAR provide a guide that enables developers to determine what subroutines are candidates for acceleration and what objects those subroutines will access for a given input set. At a more detailed level, the CIGAR methodology consists of the following steps:

1. **Preprocessing** to add annotations into the code at compile time to simplify and accelerate profile analysis done by CIGAR.
2. **Profiling** isolates compute-intensive areas of the application that we call *candidate subroutines*. We use the `gprof` profiling tool for our experiments.
3. **Data-Level Parallelism (DLP) Discovery** evaluates the amenability to coprocessor implementation of the candidate subroutine based on the amount of DLP present.
4. **Access Intensity** determines which program objects are heavily accessed by the candidate subroutines.

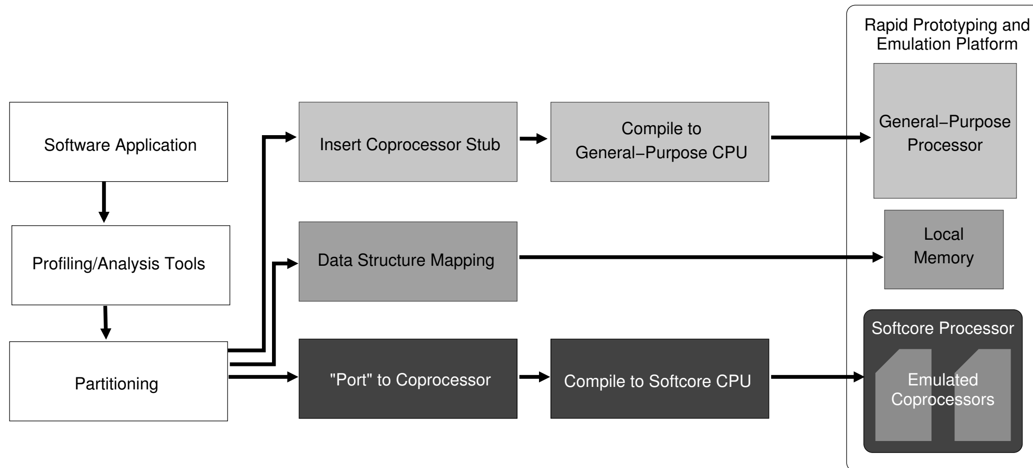

Figure 3. Design Flow Overview

5. **Liveness Analysis** measures the persistence of the data structures and to see whether it is possible for multiple objects to time-multiplex the CLM without large data transfers.
6. **Data Synchronization Granularity** maps the access pattern of the hosted data structures to one of the access models in Figure 4.

Performance profiling of the application is a well-understood filter for selecting candidate subroutines for coprocessor implementation. Due to Amdahl's Law, speedup will be limited by the time spent executing any sequential code. As such, those subroutines that dominate execution time are chosen with profiling. We set a threshold of 10% execution time before considering a subroutine for coprocessor implementation. The developer may choose to evaluate all subroutines; however, setting such a limit reduces the number of subroutines that must be tracked by CIGAR, reducing the time required for analysis.

The amount of data-level parallelism available in candidate subroutines is how we evaluate the appropriateness of implementing a candidate subroutine as a coprocessor. We evaluate loop-body DLP by counting the total number of arithmetic instructions and dividing it by the height of the dependence tree for the loop body code. Greater degrees of DLP can be exposed by exploiting cross-iteration parallelism for all of our benchmarks.

The ability of CUBA to provide demonstrable speedups depends heavily on CIGAR discovering strong correlations between data structures and code regions that can be accelerated by coprocessors. To that end, we have developed dynamic data profilers that show how much and how often data is accessed from inside candidate subroutines. We define high *access intensity* as a large number of loads and stores to a particular object in a time interval. High access intensity for a data object during a candidate routine marks it as an attractive choice for hosting in the CLM. Correlating the dynamic data profiler results with the subroutine execution periods allows the software developer to make informed decisions about what data structures should be hosted while using different candidate subroutines.

Accurate selection of data structures for inclusion in the CLM is necessary for successful coprocessor execution. However, the ability to access hosted data objects with low latency from both the CPU and coprocessor is exploited to allow for overly eager mappings to CLM. The only guarantee that must be made is that all of the data that the *coprocessor* accesses be present in its CLM prior to its execution. Data locality is enforced explicitly at the point where the processor hands off control to the coprocessor. Otherwise, since the CLM is accessible and cacheable by the CPU, correct execution will be maintained without performance degradation, even if objects are unnecessarily hosted by the CLM.

For CUBA to remain scalable and general-purpose, the architecture requires the capability of mapping large coprocessor working sets into smaller coprocessor local memories, which we will refer to as virtualization. Determining how much data is accessed from candidate subroutines allows the developer to gauge the level of CLM virtualization that must be done. For applications where the amount of data accessed is small enough, dynamic data profiling will show that the candidate objects will fit into the given CLM and no virtualization is needed. In cases where the objects are larger than available memory, software management similar to the virtualization of system memory can be employed.

As applications progress through different phases of execution, different coprocessor or data structure hosting arrangements may be appropriate. CIGAR provides a data object liveness analysis technique to aid developers in finding opportune periods to make such runtime changes. An object is said to be *dead* during the interval between a load and the first store that starts a period in which all values in that object are overwritten without an intervening load. Any dead objects can simply be discarded as soon as the last load prior to the dead interval, since it will be re-created prior to any future loads accessing its contents. Liveness is also a measure of an object's persistence in memory, with long lifetimes indicating possibly good candidates for data hosting compared to short-lived values.

Liveness analysis is useful for the situation in which a coprocessor must iterate through data structures larger than the available CLM, or when a single object is equivalent to a sequence of distinct objects. Furthermore, if the coprocessors are reconfigurable (e.g., FPGA-like) or are programmable (e.g., SPE in the Cell processor), the opportunity exists for various coprocessor functions to be utilized throughout the execution of the program. In either case, knowing that objects are dead during periods of execution allows for applications to simply discard the values in the CLM and start using the CLM to host another object. For objects that will exist later in the application, but are simply dead for a period of time, the application must change the address mapping for the object. The key feature of liveness profiling is its ability to uncover periods where objects need not be preserved, thus guiding the developer to parts of the code where coprocessor reconfiguration and

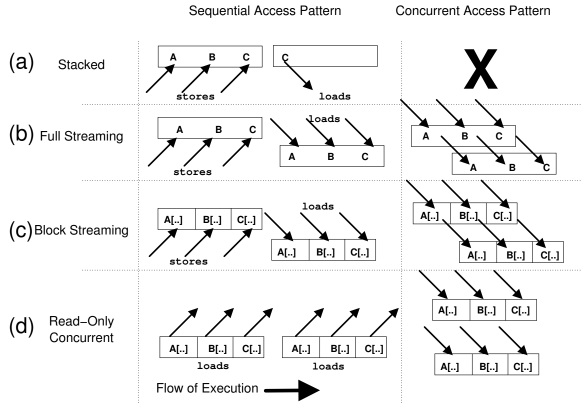

Figure 4. Data Movement in Concurrent Data Access Models

data structure remapping can take place efficiently.

The last aspect of analysis that we discuss is the granularity of data flow in a candidate subroutine. Figure 4 shows four possible modes. As part of CIGAR, we have developed a technique to analyze the data flow granularity and access behavior of our benchmarks. Figure 4(a) shows the situation where all data control is transferred at the same point. Such a model requires less architecture support, but unnecessarily limits concurrency. When the computation of the subsequent block depends on a late write by the preceding block, we call this a stack access pattern. A stack access pattern is limiting for coprocessor concurrency and if such a pattern is found, coprocessors may only be able to execute sequentially. Full streaming, as depicted in Figure 4(b), is the situation where each element is transferred independently and in an order that is consistent across control paths. In such a model, it may be possible to execute multiple coprocessors concurrently and synchronize with a mechanism similar to the presence bits used in dataflow machines [22]. Figure 4(c) shows block streaming where the access pattern between blocks is exploited, but within blocks applications may access data in an arbitrary order. Some applications have sequential code regions that only access objects in a read-only fashion, allowing for the concurrent execution shown in Figure 4(d). As part of CIGAR, we study the access patterns of applications to give direction to the developer about what computations may be able to stream data or be coscheduled in order to increase parallelism.

The tools used as part of CIGAR are developed using perl and PIN [12]. PIN tools perform the dynamic analysis of our applications by instrumenting the binaries, providing high-speed access to dynamic program behavior. We have also made the resolution of our data collection variable whenever possible to allow for faster feedback loops for developers. Using the tools provided by CIGAR, tens of millions of instructions per minute can be profiled at a fine resolution and, at coarser resolution, hundreds of millions of instructions per minute.

3.2 Emulation platform

We have developed a prototype emulation platform for CPU/coprocessor-based systems that allows *software versions of coprocessors* to be rapidly prototyped and the corresponding software to be tested and debugged without requiring developers to adopt entirely new development methods. Greater visibility is provided by our emulation platform, resulting in reduced debugging times compared to other coprocessors that rely on "black box" approaches, which do not expose their interfaces directly to the developer. By exposing the CLM to the user application, we also enable standard compiler (e.g. `gcc`) and debugging tools (e.g. `gdb`) to be used, reducing the complexity experienced by developers wishing to migrate from conventional platforms.

The emulation platform allows a developer to run applications in any of four modes that we define as: *software-only profile*, *software memory debug*, *coprocessor debug*, and *coprocessor profiling*. Using these four modes of operation, the developer can take the results of CIGAR and implement an emulated partitioning of the application. The emulated coprocessor platform can then be used to evaluate trade offs in the partitioned application's design space and to debug applications with concurrent execution on general-purpose cores and coprocessors.

The prototyping platform consists of an FPGA-based development board with embedded hard and soft processor cores that runs a fully functioning operating system and development environment. The Xilinx Virtex-II Pro FPGA [24], which integrates a PowerPC (PPC) processor core on-die, is used in this work. The software for the platform consists of Linux 2.6.18 with a full suite of GNU development tools, libraries, and utilities. The software portion of applications to be prototyped are compiled for the PowerPC processor and linked against standard PPC Linux C libraries. To emulate coprocessors, we use the Xilinx MicroBlaze [25] softcore processor running a modified form of the original source code of the region to be accelerated. The restrictions on that code are the same as those required of the coprocessors themselves, as described in Section 2.1. Furthermore, multiple coprocessors can be emulated by synthesizing multiple softcore processors, or, if they do not have overlapping lifetimes, on the same softcore processor. The hosting of application data structures is done by embedded SRAM, accessible by both the MicroBlaze and PPC processors, that is mapped into the application's address space. The embedded PPC processor, softcore MicroBlaze processors, embedded memories, and system software represent a platform that allows developers to prototype applications targeting future CPU/coprocessor systems.

We provide two modes on our emulation platform that do not use coprocessors: the software-only mode and the software memory debug mode. The software-only mode is the original software application running on the emulation platform. Applications can be developed using conventional compilation and debugging tools under the software-only mode to ensure a stable base prior to partitioning into coprocessor and general-purpose software modules. The software memory debug mode executes all of the code on the general-purpose processor, however, the data structures to be hosted by the CLM are allocated to the embedded memories of the emulation platform. Placing the selected objects into the CLM allows for initial debugging of the software/coprocessor interface and an initial analysis of the caching and bus contention behavior of the final design.

The coprocessor debug and profiling modes are used post-partitioning to debug and evaluate the partitioned designs, respectively. In the coprocessor debug mode, emulated coprocessors are used that are functionally identical to the final coprocessor design, however, they run as a software module on a softcore processor. The coprocessor debug mode allows the developer to remove bugs in the synchronization and data partitioning between the CPU and coprocessor. For applications that have many objects needing to be hosted and for virtualization of the CLM, ensuring that the accelerator local memory contains the correct set of values is critical for correct execution. Having the ability to verify the partitioning scheme using emulation as opposed to simulation can reduce

debugging time by two orders of magnitude, as our results show (see Table 2). Furthermore, our platform provides a high degree of visibility and control for the developer to alter and examine the state of the executing application using symbolic debuggers and external interfaces via JTAG.

The coprocessor profile mode is the final step in the design flow, removing the emulated coprocessor from the hardware interface and inserting the actual coprocessor. Doing so enables the developer to evaluate the performance of the accelerated and *debugged* design. We demonstrate CIGAR as a methodology for partitioning applications and leave performance analysis to future work. Using these two modes, the developer can remove bugs and evaluate performance—in that order—allowing for rapid debugging of the software/hardware interface followed by performance evaluation.

3.3 Summary

We present an overview of the dynamic profiling methodology, rapid prototyping environment, and coprocessor emulation platform, that together make up the CIGAR design flow, which were developed to aid in the process of mapping applications on to CPU/coprocessor architectures.

We presented four modes of operation for our CPU/coprocessor emulation platform. The platform enables rapid prototyping of partitioned designs by using softcore processors and FPGA logic to emulate otherwise unavailable coprocessors. The goal of these modes is to defer the most costly steps in the prototyping process and to accelerate exploration of the design options to convergence on a *correct* design faster than is possible through simulation. The emulation platform described enables this acceleration by enabling design space exploration prior to the availability of the target coprocessor hardware.

4. Design Examples

We present the application which drove our tool development, along with the tools themselves, followed by two case studies of application partitioning using CIGAR. We draw examples from the SPECint2006 suite in three application domains characterized by computation amenable to coprocessor acceleration: mathematical libraries and simulation (462.libquantum), scientific computing (456.hmmer), and multimedia (464.h264ref). Finally, we demonstrate the utility of a rapid prototyping platform that allows software developers to target future hybrid CPU/coprocessor architectures with their applications.

4.1 Design driver: 462.libquantum

The driving application used to develop our partitioning infrastructure is the 462.libquantum integer benchmark. The benchmark was chosen for its small code size and easily discoverable regions with data-level parallelism (DLP), making it an easy testbed for our analysis techniques. The benchmark simulates a quantum computer running Shor's algorithm for integer factorization relying heavily upon the libquantum library. On our emulation platform, we migrate two library subroutines and their associated data into emulated coprocessors. The benchmark spends over $3/4$ of the original uniprocessor runtime in these two functions. We provide examples of how our profile tools have enabled us to determine the data structures appropriate for coprocessor local memory hosting. We then show how such an application is instrumented to work for our different platform computation modes. We further explain how debugging in our model proceeds, resulting in a semantically correct version of the partitioned application.

After applying CIGAR, the developer has a set of subroutines amenable to coprocessor implementation, the data objects to be hosted by those coprocessors, knowledge of the persistence and periods of liveness of those data objects. The current implementation of the tools only provides information, leaving all code modification and final inspection to the developer. Greater automation of partitioning is desireable, but due to its inherent complications, we leave automation to future work. We now present the process that led us to developing CIGAR using 462.libquantum as an illustrative example.

Preprocessing: We have developed a set of scripts that add annotations to the source to simplify and expedite the profiling and analysis process. For example, this step adds tags indicating to the tools the names of dynamically allocated objects. Static analysis and correctness checks in this stage could provide feedback and semantic information for the subsequent steps, but are not investigated here.

Profiling: Application profiling is used to determine candidate subroutines based on their contribution to the overall computation time. For 462.libquantum we find that 52% and 26% of the computation time is spent in the subroutines `quantum_toffoli()` and `quantum_sigma_x()`, respectively, making them good targets for acceleration.

DLP Discovery: Our chosen SPEC benchmarks are tuned for sequential execution and thus require some code transformations to expose cross-iteration loop parallelism. Table 1 shows the DLP our method found within loop bodies. We do not have an explicit technique for choosing a method for exposing more parallelism, but after inspection of the DLP regions found during DLP discovery, we found simple transformations could expose more DLP and are shown in the table. For CIGAR, we calculate the DLP present in loops of candidate subroutines by comparing the store set of each iteration with the load set of subsequent iterations. If there is a cross-iteration loop read-after-write dependence, we declare that the loop is not data-parallel and therefore not amenable to coprocessor implementation.

Access Intensity: CIGAR provides the ability to visualize the correlation between different data objects and candidate subroutines. Figure 5 shows three candidate subroutines for the libquantum benchmark and two candidate data objects, `reg->node` and `reg`. The width of the data structure line is proportional to the number of memory accesses performed in a time interval, or the access intensity. During the interval depicted, as is true for much of the application, the access intensity of these objects is far greater than of any other objects, and in fact the three functions shown do not access any other data objects. The peak access intensity is one in nine instructions being either a load or a store to the given object. Furthermore, even in the regions between the execution of candidate subroutines, where the CPU would be accessing the data, we find moderate access intensity for candidate data objects. Our model exploits this access pattern by providing unequal sharing of the structures hosted in the CLM, optimizing for the more frequent coprocessor data accesses, but allowing cacheable CPU access while the coprocessor is not executing.

Liveness Analysis: The Access Intensity stage shows there to be a high correlation between accesses to certain data objects and the candidate subroutines we have identified. Being able to measure the lifetime of these objects provides information necessary for the developer to decide whether a pass-by-reference or a pass-by-value model is appropriate. Our liveness technique, a scriptable combination of profiling and instrumentation, demonstrates that for 462.libquantum, the candidate objects are persistent, i.e., they stay live for the duration of the application's execution, and

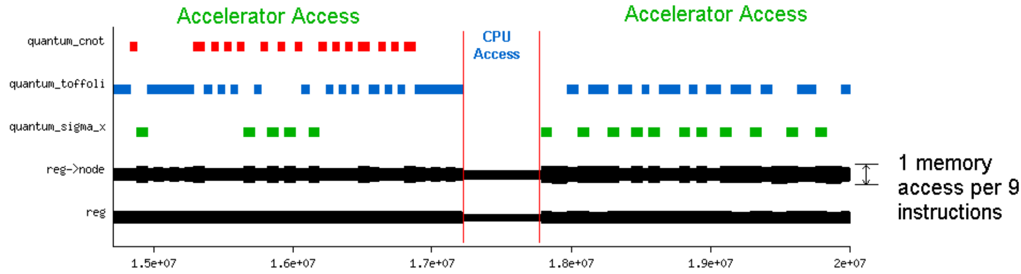

Figure 5. Memory access intensity for 462.libquantum. The top three lines indicate which function is executing at each point in time.

are therefore amenable to hosting throughout the application's execution. Furthermore, since the CLM is limited in size and hosted data is not backed by system memory, liveness analysis can be used to identify data that need not be copied back to memory upon remapping of the CLM.

Data Synchronization Granularity: We have investigated the required granularity of data synchronization for 462.libquantum. Having coarse-grained data synchronization allows for a reduced complexity architecture and programming model where the transfer of data object control between processing elements occurs in whole-object transactions as shown in Figure 4(a). However simple, such a coarse-grained model may result in limited concurrency since one operation must completely relinquish control of an object before the next operation can access it. A more fine-grained approach would result in added concurrency when the results of one operation are forwarded to the next (Figure 4(b)), essentially pipelining multiple operations inside the coprocessor. However, the added synchronization may result in unjustifiable overhead compared to a coarse-grained mechanism.

Visual inspection of 462.libquantum found that coprocessor operations could be overlapped. Our analysis using CIGAR has shown this to be the often the case with the data produced by `quantum_sigma_x()` available to `quantum_toffoli()` when the calls are adjacent, allowing for a pipelined implementation. However, due to limitations of our prototype emulation platform, we are unable to demonstrate concurrent emulated coprocessors.

Using the approach developed in the design driver we now present case studies of benchmarks evaluated using the CIGAR infrastructure.

4.2 Case study: 456.hmmer

The 456.hmmer benchmark is a DNA sequencing application that uses Hidden Markov Models to perform DNA sequence alignments. At the thread level, the benchmark can be partitioned such that independent threads each process independent blocks of data from an assigned work pool. Within each thread, the benchmark's core contains exploitable DLP. We apply CIGAR to uncover the DLP, map it into CUBA, and prototype it in our emulation platform.

Profiling indicates that 456.hmmer spends between 71% and 97% of its execution time in the `P7Viterbi()` subroutine. Our CIGAR analysis indicates that `P7Viterbi()`'s inner loop accesses four data objects heavily, all of which are arrays: `mx->imx_mem`, `mx->mmx_mem`, `mx->dmx_mem`, and `mx->xmx_mem`. Liveness analysis provided us with Figure 6, demonstrating that these data objects all become dead early in each invocation of `P7Viterbi()`, and some of the objects are occasionally dead between invocations as well. This information led us to further inspect the code and to notice that data-structure resizing occurs in `ResizePlan7Matrix()`, called from `P7Viterbi()` before executing the compute-intense code portions. We discovered that a call to the C library function `realloc()` is the source of loads from the objects that extend the liveness intervals beyond `P7Viterbi()`. Thus, between invocations to `P7Viterbi()`, the objects contain no useful state, indicating that there is no need to transfer the arrays into or out of the accelerator between invocations. Additionally, the deadness of the arrays indicates that coprocessor context switching overhead is minimal between invocations.

We have implemented 456.hmmer on our prototype emulation platform, hosting the four candidate data objects in the CLM and the inner loops of the `P7Viterbi()` emulated on the softcore processor. The CIGAR infrastructure has allowed us to arrive at this partitioning and our emulation platform has allowed us to rapidly prototype the design without having a physical coprocessor implemented.

4.3 Case study: 464.h264ref

The 464.h264ref benchmark is a multimedia application that encodes raw video into a compressed form using the H.264 specification. The benchmark is a streaming application which applies various transformations to blocks of data as they are read from the input video and are incorporated into the encoded output. We isolate one stage in the encoding process using CIGAR and implement it as an emulated coprocessor.

Profiling indicates that two subroutines, `SetupFastFullPel Search()` and one of its children `SATD()`, account for roughly 45% of 464.h264ref's execution time. `SATD()` computes the sum of absolute differences (SAD) over an array passed to it by `Setup-Fast FullPelSearch()`. The input array represents the current block (`SAD_block`), composed by pulling pointers from an array representing the current video frame.

Our DLP tools were able to discover that `SATD()` invocations are independent and that the SAD computation has a high level of instruction-level parallelism. Thus `SATD()` is amenable to coprocessor acceleration. Our liveness analysis showed that `SAD_block` is live upon `SATD()` invocation, but dead when `SATD()` returns, indicating that the function returns its computational result only via its return value. Therefore `SATD()` does not modify its input, and the coprocessor need not transfer `SAD_block` back to the CPU after execution. Thus, we map the SAD kernel into an emulated coprocessor, hosting the current `SAD_block` in the CLM.

The `SATD()` subroutine has a simple if-then-else structure that would require added resources or added complexity if implemented

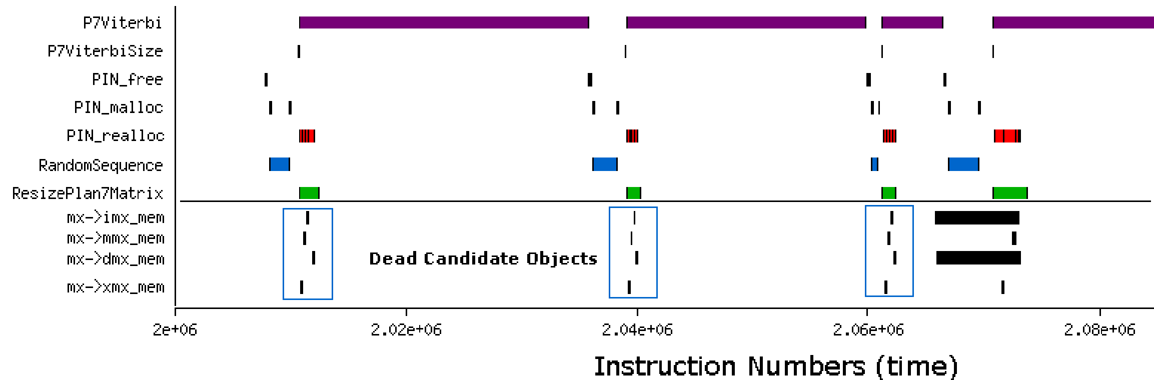

Figure 6. Liveness results for 456.hmmer with horizontal bars for data indicating dead regions. (Note: Due to the resolution of the image, function invocations appear to overlap.)

as a coprocessor. Profiling indicated that for our input sets, the branch is highly biased and we are able to implement a data-parallel emulated coprocessor excluding the uncommon control path. Accounting for all possible execution paths involves a trade-off between a more complicated coprocessor and the overhead of the exception handling mode.

4.4 Emulation platform evaluation

Our coprocessor emulation platform has been developed to enable software developers to rapidly prototype applications partitioned using the CIGAR analysis infrastructure. As an example of its use, we isolated two kernels of code from the 462.libquantum benchmark that we execute on a softcore processor. The emulated coprocessor runs independently of the CPU, with synchronization performed via memory-mapped registers and explicit cache flushes. Two of the data structures isolated using CIGAR were mapped into the CLM of the emulated coprocessor, with that memory being accessible to both the CPU and the coprocessor. The only changes that were made to the application were to allocate the memory used by the mapped objects to the CLM and to perform synchronization at the call site of the now accelerated subroutine. Following this same flow, we have additionally implemented 456.hmmer and 464.h264ref on our emulation platform based on the partitioning found using CIGAR.

A strong motivation for using softcore coprocessor emulation is that our platform executes the partitioned application much closer to its original speed than does a software-only system simulator. Table 2 demonstrates the time required to run the SPEC test input sets for our benchmarks. The first row of Table 2, labeled ‘Native Execution’, is the time required to run on a 3.2 GHz Intel Pentium 4, representing a contemporary system. Note that this result does not include CIGAR instrumentation present and represents best case performance for the machine. The next row compares the execution time of our platform running the code solely in software on the embedded hard processor of our platform. The next two rows provide the results of using emulated coprocessors with the benchmark mostly running on the embedded hard processor and the coprocessor running as software on a softcore processor. For each of these, we show emulated coprocessors with their hosted data structures cached and not cached. The last row of the table shows the time to simulate the benchmarks using a cycle-accurate simulator we are developing for CUBA.

CIGAR instrumentation occurs on the native platform, providing the partitioning information quickly. The emulated coprocessor prototype platform provides a means of developing, debugging, and evaluating a partitioned design. Non-cached local store access simplifies debugging, since it allows the developer to stop execution and view a coherent global state, including local memories, system memory, the register state of the general-purpose processor, utilizing the pre-existing software tools for the FPGA platform. Having this degree of visibility while suffering less than a two order of magnitude slowdown in performance allows for faster, easier development of partitioned applications. To more realistically evaluate a partitioned design, caching can be enabled. Once a partitioned design is debugged, performance evaluation of a *correct* design can be carried out on a simulator.

While one to two orders of magnitude slower than native execution, the emulation platform allows for incremental mapping of applications into a prototype of CUBA, without suffering the three to four orders of magnitude slowdown experienced when using cycle-accurate simulation. The speed of the emulation platform allows developers to partition their applications using CIGAR and to debug them effectively without suffering the high turnaround time associated with simulation or full-blown coprocessor generation. To contrast the design effort of writing a small amount of code for the emulated coprocessor versus a real-world implementation, see [13] for HMMer and [21] for motion estimation. When the flexibility of a simulation platform is needed, the developer can move his now-debugged design onto the simulator for evaluation. Future research could leverage performance characteristics tracked using the cycle-accurate simulator to better model CUBA on the emulation platform, providing the developer with both the speed of emulation and the accuracy and flexibility of the simulator.

5. Conclusion

CUBA is presented as an prototype architecture for general-purpose processor systems that incorporate closely-coupled coprocessor resources that have local memories that host data structures of applications. The main contribution of this work is CIGAR, a methodology for mapping applications into heterogeneous CPU-/coprocessor architectures such as CUBA. We provide examples of general-purpose applications that were mapped into a CPU/coprocessor emulation platform. The emulation platform allows for

	462.libquantum		456.hmmer		464.h264ref	
Native Execution	1x	(0.30s)	1x	(0.10s)	1x	(1m13s)
Emulation Platform w/o Accel	40x	(12.1s)	43x	(4.33s)	26x	(32m11s)
Emulation w/Accel+NoCache	60x	(18.1s)	71x	(7.10)	30x	(36m23s)
Emulation w/Accel+Cache	56x	(16.7s)	73x	(7.33)	30x	(36m26s)
Simulation w/o Accel	2437x	(12m11s)	1180x	(1m58s)	3151x	(3833m50s)

Table 2. Slowdown for alternate execution modes with example applications.

software designers to partition their software applications across coprocessor and general-purpose processor domains. Using the presented design flow and tools, developers can rapidly prototype software that targets heterogeneous systems incorporating general-purpose processors and application-specific coprocessors.

Acknowledgements

The authors acknowledge the support of the Gigascale Systems Research Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program. This work has been supported by the European Commission under the HiPEAC NOE (IST-004408) project, and the Spanish Ministry of Education CICYT TIN-2004-07739-C02-01. We also thank Xilinx and Intel for their generous donations. The authors would also like to thank Shane Ryoo for his insightful comments.

References

[1] ASANOVIC, K., AND ET AL. The landscape of parallel computing research: A view from berkeley. Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] GONZALEZ, R. E. A software-configurable processor architecture. *IEEE Micro 26*, 5 (2006), 42–51.

[3] GSCHWIND, M., AND ET AL. Synergistic processing in cell's multicore architecture. *IEEE Micro 26*, 2 (2006), 10–24.

[4] HAUSER, J. R., AND WAWRYZNEK, J. Garp: A MIPS processor with a reconfigurable coprocessor. In *IEEE Symposium on FPGAs for Custom Computing Machines* (1997).

[5] HERLIHY, M., AND MOSS, J. E. B. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture* (New York, NY, USA, 1993), ACM Press, pp. 289–300.

[6] JACOB, J. A., AND CHOW, P. Memory interfacing and instruction specification for reconfigurable processors. In *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays* (New York, NY, USA, 1999), ACM Press, pp. 145–154.

[7] KARURI, K., AND ET AL. Fine-grained application source code profiling for ASIP design. In *DAC '05: Proceedings of the 42nd annual conference on Design automation* (New York, NY, USA, 2005), ACM Press, pp. 329–334.

[8] KNIGHT, T. F. An architecture for mostly functional languages. In *Proceedings of ACM Lisp and Functional Programming Conference*. Aug 1986, pp. 500–519.

[9] KONGETIRA, P., AINGARAN, K., AND OLUKOTUN, K. Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro 25*, 2 (2005), 21–29.

[10] KUMAR, R., AND ET AL. Single-ISA heterogeneous multicore architectures: The potential for processor power reduction. *MICRO* (2003), 81–92.

[11] KUMAR, R., AND ET AL. Single-ISA heterogeneous multicore architectures for multithreaded workload performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA04)* (2004).

[12] LUK, C.-K., AND ET AL. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2005), ACM Press, pp. 190–200.

[13] MADDIMSETTY, R. P., BUHLER, J., CHAMBERLAIN, R. D., FRANKLIN, M. A., AND HARRIS, B. Accelerator design for protein sequence HMM search. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing* (New York, NY, USA, 2006), ACM Press, pp. 288–296.

[14] RAMANATHAN, R. Extending the world's most popular processor architecture. Whitepaper, Intel, September 2006.

[15] SALAPURA, V., AND ET AL. Power and performance optimization at the system level. In *CF '05: Proceedings of the 2nd conference on Computing frontiers* (New York, NY, USA, 2005), ACM Press, pp. 125–132.

[16] SANKARALINGAM, K., KECKLER, S. W., MARK, W. R., AND BURGER, D. Universal mechanisms for data-parallel architectures. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2003), IEEE Computer Society, p. 303.

[17] SINGH, H., AND ET AL. MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Transactions on Computers 49*, 5 (2000), 465–481.

[18] SINHAROY, B., AND ET AL. Power5 system microarchitecture. *IBM J. Res. Dev. 49*, 4/5 (2005), 505–521.

[19] STAFF, N. *NVIDIA CUDA: Programming Guide*, 0.8 ed. NVIDIA, February 2007.

[20] SURESH, D. C., AND ET AL. Profiling tools for hardware/software partitioning of embedded applications. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems* (New York, NY, USA, 2003), ACM Press, pp. 189–198.

[21] VASSILIADJS, S., HAKKENNES, E., WONG, J., AND PECHANEK, G. The sum-absolute-difference motion estimation accelerator. In *Proceedings of Euromicro Conference* (1998).

[22] VEEN, A. H. Dataflow machine architecture. *ACM Comput. Surv. 18*, 4 (1986), 365–396.

[23] WITTIG, R., AND CHOW, P. OneChip: An FPGA processor with reconfigurable logic. In *IEEE Symposium on FPGAs for Custom Computing Machines* (Los Alamitos, CA, 1996), K. L. Pocek and J. Arnold, Eds., IEEE Computer Society Press, pp. 126–135.

[24] XILINX STAFF. *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, October 2005.

[25] XILINX STAFF. *MicroBlaze Processor Reference Guide*, 6.0 ed. Xilinx, June 2006.