Bahurupi: A Polymorphic Heterogeneous Multi-Core Architecture

MIHAI PRICOPI and TULIKA MITRA, National University of Singapore

Computing systems have made an irreversible transition towards parallel architectures with the emergence of multi-cores. Moreover, power and thermal limits in embedded systems mandate the deployment of many simpler cores rather than a few complex cores on chip. Consumer electronic devices, on the other hand, need to support an ever-changing set of diverse applications with varying performance demands. While some applications can benefit from thread-level parallelism offered by multi-core solutions, there still exist a large number of applications with substantial amount of sequential code. The sequential programs suffer from limited exploitation of instruction-level parallelism in simple cores. We propose a reconfigurable multi-core architecture, called Bahurupi, that can successfully reconcile the conflicting demands of instruction-level and thread-level parallelism. Bahurupi can accelerate the performance of serial code by dynamically forming coalition of two or more simple cores to offer increased instruction-level parallelism. In particular, Bahurupi can efficiently merge 2-4 simple 2-way out-of-order cores to reach or even surpass the performance of more complex and power-hungry 4-way or 8-way out-of-order core. Compared to baseline 2-way core, quad-core Bahurupi achieves up to 5.61 speedup (average 4.08 speedup) for embedded workloads. On an average, quad-core Bahurupi achieves 17% performance improvement and 43% improvement in energy consumption compared to 8-way out-of-order baseline core on a diverse set of embedded benchmark applications.

Categories and Subject Descriptors: C.1 [Computer System Organization]: Processor Architectures

General Terms: Design, Performance

Additional Key Words and Phrases: Instruction-level parallelism, thread-level parallelism, multi-core.

ACM Reference Format:

Pricopi, M. and Mitra, T. 2012. Bahurupi: A polymorphic heterogeneous multi-core architecture. *ACM Trans. Architect. Code Optim.* 8, 4, Article 22 (January 2012), 21 pages.
DOI = 10.1145/2086696.2086701 <http://doi.acm.org/10.1145/2086696.2086701>

1. INTRODUCTION

The distinction between high-performance embedded architecture and general-purpose computing architecture is rapidly disappearing especially in the consumer electronics domain. A smartphone today is expected to support a very dynamic and diverse landscape of software applications. Traditional multi-core architecture, that is just a collection of identical simple cores, is not suitable for this kind of workload. On the other hand, embedded systems generally embrace heterogeneous multi-core solutions customized for a particular application domain that can offer significant advantage in terms of performance, power, and area [Jerraya and Wolf 2005]. While such customizations are beneficial for specific kernels such as audio, video, image processing, high-performance embedded platforms (e.g., smartphones) need to execute a wide variety of general-purpose applications for which the workload is not known a-priori.

This work was partially supported by Singapore Ministry of Education Academic Research Fund Tier 1 MOE2009-T2-1-033.

Authors' address: M. Pricopi and T. Mitra, Computer Science Department, School of Computing, National University of Singapore; email: {mihai, tulika}@comp.nus.edu.sg.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2012 ACM 1544-3566/2012/01-ART22 \$10.00

DOI 10.1145/2086696.2086701 <http://doi.acm.org/10.1145/2086696.2086701>

Fig. 1. Bahurupi architecture. Additional resources required for coalition are highlighted.

In this paper, we propose a polymorphic heterogenous multi-core architecture, called Bahurupi (an Indian word meaning a person of many forms and guides, a polymorph), that can be tailored according to the workload by software. Bahurupi is fabricated as a homogeneous multi-core system containing multiple identical, simple cores. The main novelty of Bahurupi lies in its ability to morph itself into a heterogenous multi-core architecture at runtime under software directives. Post-fabrication, software can compose together the primitive cores to create a customized multi-core system that best matches the needs of the applications currently executing on the system.

Bahurupi successfully re-conciliates the conflicting requirements of applications with explicit thread-level parallelism (TLP) and single-threaded serial applications with high degree of instruction-level parallelism (ILP). Bahurupi architecture comprised of multiple simple homogeneous cores is ideally suited to take advantage of TLP. But there exists a large class of applications with substantial sequential code fragment that are difficult, if not impossible, to parallelize. Amdahl's law states that the speedup of such applications will be limited by the performance of the serial code. Only complex out-of-order execution engines with high-degree of superscalarity (4-way or 8-way) can transparently accelerate sequential code fragments through aggressive exploitation of ILP. Power and thermal limits as well as reliability issues, however, does not permit deployment of such complex cores in embedded systems. [Hill and Marty 2008] showed through simple analysis how an application can achieve speedup much beyond the limit of Amdahl's law if the underlying architecture can be dynamically reconfigured between small number of complex core exploiting ILP and large number of simple cores exploiting TLP. Bahurupi achieves this seamless transition between ILP and TLP. At runtime, Bahurupi can form coalition of 2-4 cores to create virtual 4-way or 8-way superscalar cores.

Bahurupi Architecture. Figure 1 depicts a high-level overview of Bahurupi architecture. Each core is a simple 2-way out-of-order (ooo) processor. Four such simple cores form a cluster. The example architecture in Figure 1 consists of 2 such clusters; but the number of clusters can be easily scaled with technology. In normal mode, this multi-core architecture can efficiently support multi-threaded execution. The simple 2-way cores, however, cannot exploit much ILP in sequential applications. To solve this issue, Bahurupi can form dynamic coalition of 2-4 cores in a cluster with minimal additional

Fig. 2. Speedup trends for Ferret kernels and overall speedup with reconfigurable architecture.

hardware such that the merged cores can substantially speed up serial code execution. When 2 cores (4 cores) form a coalition, Bahurupi achieves performance close to that of 4-way (8-way) ooo execution engine. Note that Bahurupi allows coalition of at most 4 cores as speedup is limited beyond 8-way ooo core. The figure shows the architecture running four applications—one high-ILP serial thread running on 2-core coalition, two low-ILP serial threads running on two independent cores and one very high-ILP serial code running on 4-core coalition. In summary, Bahurupi can achieve the performance of complex ooo superscalar processor without paying the price of complex hardware and its associated energy inefficiency and reliability issues.

Motivating Example. As a concrete motivating example, we present here a case study of *Ferret* benchmark from PARSEC suite [Bienia et al. 2008] that performs image similarity search. The application consists of six kernels. We first run the sequential version of the application on one complex core configured as 2-way, 4-way, and 8-way ooo execution engines, respectively, using MARSS [Patel et al. 2011] simulator. We collect execution time for each individual kernel. Next we create 2-core and 4-core homogeneous multi-core systems where each core is a 2-way ooo engine. For 2-core system (4-core system), we create 2 threads (4 threads) for each of the kernels except for *load* and *out*, which are hard to parallelize. Now we run this multi-threaded application on multi-core and collect execution time for each kernel. Figure 2 shows the speedup trend for each kernel as we increase ILP and TLP normalized with respect to 2-way core. *seg* and *rank* can benefit from TLP while *extract* and *vec* benefit from ILP.

For the whole application, the speedup from static homogeneous configurations (4-way ooo, 8-way ooo, 2x2-way core, and 4x2-way core) are shown on the right. None of these configurations can exploit both ILP and TLP. A 4-core dynamically reconfigurable architecture like Bahurupi can run *seg* and *rank* on 4x2-way cores to exploit TLP, while the rest of the kernels can be run on virtual 8-way ooo engine by forming coalition of 4 cores. Thus the 4x2-way reconfigurable architecture improves the speedup by 38% compared to static homogeneous 4x2-way cores. This case study confirms that dynamically reconfigurable architectures, such as Bahurupi, that can seamlessly transition between ILP and TLP can provide significant performance boost to applications.

The idea of composing simple cores together to accelerate serial code has been explored before. However, such solutions either require significant additional shared hardware resources [Kumar et al. 2004a] and modifications to the internal architecture of the composable cores [Ipek et al. 2007] or follow a radically different instruction-set architecture (ISA) that require complete re-engineering of the software model and aggressive compiler optimizations [Kim et al. 2007]. Bahurupi, in contrast, is a hardware-software cooperative solution that demands minimal changes to both hardware and software. In coalition mode, Bahurupi follows a distributed execution model that avoids complex centralized fetch/decode, dependency resolution, and instruction scheduling of previous approaches. It needs support from compiler to identify the basic

blocks and their register dependencies. This information is encapsulated in special sentinel instructions that precede each basic block. Thus the dependencies among the basic blocks are explicitly conveyed in the program code. The cores can fetch and execute different basic blocks in parallel, thereby achieving performance speedup. The only additional hardware required is the coalition logic as shown in Figure 1. The coalition logic includes a shared global register file to communicate values between basic blocks running on different cores as well as three dedicated registers for synchronization among the cores. A minimal amount of additional resources is added to the internal architecture of the cores to support coalition. The main execution engines of the cores remain completely unchanged.

Compared to a 2-way core, dual-core Bahurupi can achieve up to 3.24 times speedup (average speedup 2.64) while quad-core Bahurupi can achieve up to 5.61 times speedup (average speedup 4.08) for embedded applications. More importantly, dual-core and quad-core Bahurupi can even surpass the performance of significantly more complex 4-way and 8-way cores, respectively, for a large range of applications.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 presents the execution model of Bahurupi. Section 4 describes in detail the novel architecture. Section 5 presents the methodology that we used in our evaluation and describes the results. Section 6 concludes.

2. RELATED WORK

Some previously proposed architectures have attempted to adapt multi-cores to speed up sequential applications. Kumar et al. [2004b] proposed an asymmetric chip multiprocessor that comprises of cores with different size and performance. The advantage of this architecture is low power consumption and high performance achieved by dynamically moving programs from one core to another in order to reach an optimal point. Besides lack of flexibility, this architecture introduces high degree of unpredictability for software applications [Balakrishnan et al. 2005]. Thus, it is desirable to use similar cores and merge them together.

The Core Fusion architecture [Ipek et al. 2007] can fuse homogeneous cores to improve single-thread performance. Compared to Bahurupi, Core Fusion uses more complex distributed hardware leading to higher performance overhead. In addition, Bahurupi requires minimal modification to internal micro-architecture of the cores, whereas Core Fusion uses collective fetch management and register renaming thereby completely bypassing internal core renaming logic. Being the closest to our architecture, this work reports an average speedup of 50% for the floating-point SPEC applications and 30% for the integer SPEC applications when using a quad-core fused configuration compared with fine-grain 2-way CMP. In contrast with Core Fusion, quad-core Bahurupi obtains an average speedup of 91% for SPEC integer applications and 210% for SPEC floating-point applications compared to baseline 2-way core.

TFlex [Kim et al. 2007] uses no physically shared resources to merge its composable lightweight processors. Instead, it relies on a special distributed microarchitecture (EDGE) that is configured to implement composable lightweight processors. Bahurupi, in contrast, can be implemented on top of conventional CISC and RISC architectures. EDGE ISA also groups instructions into blocks which is similar to our sentinel instructions. However, EDGE ISA adds complex bookkeeping to the block structures and relies on point-to-point communication to exchange information among cores.

Conjoining in-order processors to form complex ooo cores introduces fundamental obstacles limiting achievable performance [Salverda and Zilles 2008]. Simply fusing in-order cores does not bring any performance increase if the cores do not have minimal out-of-order capabilities. To overcome this issue, Federation [Tarjan et al. 2008] merges

a pair of scalar cores to create 2-way out-of-order (ooo) cores by adding substantial ooo logic to the internal pipeline stages.

Voltron [Zhong et al. 2007] uses multiple homogeneous cores that can be adapted for single and multi-threaded applications. The cores can operate in coupled mode where they act as a VLIW processor to exploit the hybrid forms of parallelism found in the code. Voltron relies on a very complex compiler that exploits parallelism from serial code by partitioning the code into small threads, scheduling the instruction to the cores and directing the communication among the cores.

The idea of using special instructions to carry liveness information among blocks of code was explored in Vajapeyam et al. [2008]. Using trace descriptors, this design allows processors go beyond basic block limits in program order. This architecture uses a shared ROB, a shared fetch unit and very simple execution units (that have only fetch, issue and execute stages) instead of ordinary cores.

An asymmetric chip multiprocessors alternative is presented in Suleman et al. [2009]. This overcomes the serialization effect of a critical section for multithreaded application where threads do not usually finish their jobs at the same time. But the synchronization mechanisms can make a considerable number of fast threads wait for the slow threads. This work proposes that architectures should have a dedicated powerful core that would help accelerate the slow threads in order to reach their deadline faster. Our design is a better alternative as it does not need a dedicated powerful core in order to accelerate the thread. If the scheduler decides that a thread must be accelerated then Bahurupi will form a bigger coalition to run the thread.

In Etsion et al. [2010], dynamically forked tasks are executed in an out-of-order fashion by cores behaving as functional units. The dependency information between tasks is explicitly defined by the programmer in a special programming model. Complex hardware handles decoding, building task-graph and scheduling tasks to the cores. This work focuses on improving task-level parallelism while Bahurupi is designed to accelerate sequential code using simple cores. Our design does not need a special programming model and the dependency information between basic blocks is encoded by the compiler which needs minor modifications.

As Bahurupi does not need a special execution model or ISA, it is distinguished from other works: TRIPS [Sankaralingam et al. 2003] and speculative architectures such as Multiscalar [Sohi et al. 1995].

3. BAHURUPI EXECUTION MODEL

Bahurupi reconfigurable multi-core architecture allows cores to form coalitions so as to improve single-thread performance. A *coalition* is defined as a group of cores working together to accelerate the execution of a serial stream of instructions. In normal mode, Bahurupi executes multi-threaded application on a set of homogeneous cores. One simple core might not be powerful enough to exploit the amount of ILP [Wall 1991] available in some threads. In that scenario, Bahurupi architecture configures a subset of its cores to run in coalition mode so that the virtual cores can extract more ILP and implicitly execute the threads faster. The design uses limited amount of additional hardware that is shared among the cores and minimal compiler modifications.

Bahurupi architecture follows a distributed execution model in coalition mode. The unit of execution for a core is a basic block—a sequence of instructions with single entry and single exit point. A core fetches and executes one basic block of instructions at a time. Our goal is to execute the basic blocks in parallel on the cores that form coalition and thereby achieve speedup for serial code.

Bahurupi execution model is similar to thread pool pattern in parallel computing where a number of threads are created to execute a number of tasks. The number of tasks is usually much more compared to the number of threads. As soon as a thread

Fig. 3. The sentinel instruction format.

Fig. 4. Percentage of basic blocks with number of live-in and live-out registers below a threshold.

completes its task, it requests the next ready task until all the tasks have been completed. In Bahurupi architecture the cores correspond to the threads and the basic blocks correspond to the tasks. The cores fetch basic blocks for execution. As soon as a core completes fetch, decode, and rename of all the instructions in a basic block, it attempts to fetch the next available basic block.

To achieve the execution model of Bahurupi, we need to first resolve register and memory dependencies among the basic blocks so as to maintain correctness during parallel execution. Instead of relying on a complex hardware mechanism to detect inter-dependency among the basic blocks, we resort to a hardware-software co-operative solution. Second, we need to ensure that the cores fetch, rename and commit the basic blocks in program order. The execution of the instructions from different basic blocks can be performed out-of-order and the main speedup of Bahurupi comes from this out-of-order parallel execution of instructions from different basic blocks on different cores.

3.1. Sentinel Instruction

We let the compiler detect the live-in and live-out registers corresponding to each basic block. The term, *live-in register*, indicates a register that is alive at the entry of a basic block and is actually used inside the basic block. The term, *live-out register*, stands for a register that is alive at the exit of a basic block and is actually updated inside the basic block. The live-in and live-out registers correspond to inter basic block dependencies. It is possible for the same register to appear as both live-in and live-out register. Now we need to communicate the live-in, live-out information to the hardware architecture. We introduce a new instruction, called *sentinel*, to encode this information. The compiler adds a sentinel instruction in the beginning of each basic block. That is, the compiler splits the program into basic blocks which are delimited by sentinel instructions.

Bahurupi design can be applied to any ISA that can be extended with the sentinel instruction. Figure 3 depicts the format of a sentinel instruction. Our design assumes 64-bit instruction format. The BB_SIZE field specifies the length of the basic block which is delimited by this sentinel instruction. We set this field to 4 bits, that is, we can support at most 16 instructions in a basic block. If the size of a basic block exceeds 16 instructions, it is split into two or more basic blocks. The BB_TYPE is a 1-bit field that specifies if the basic block ends with a branch instruction or not.

The next six fields hold the live-in and live-out registers for the basic block. We decide to use three live-in and three live-out registers after evaluating multiple benchmarks. Figure 4 depicts the percentage of basic blocks with number of live-in and live-out

registers below certain thresholds for some SPEC and embedded benchmarks. For almost all benchmarks, 90% of basic blocks contain less than or equal to 3 live-in and live-out registers. If a basic block contains more than 3 live-in or live-out registers, the compiler splits the basic block into two or more sub-blocks so as to satisfy the constraint. The size of a live-in and live-out field is 6 bits for an ISA with 32 integer registers and 32 floating point registers. If an ISA does not offer enough bits to encode the live-in and live-out registers, we can set aside a limited number of registers as global live-in, live-out registers. This is similar in spirit to using specific registers to transfer parameters during subroutine calls. The compiler has to perform register swapping to map all live-in, live-out registers to these specific set of registers. We then only need to encode which subset of registers from this specific set has been used in a particular basic block.

Figure 6 shows the sentinel instruction for a basic block with five instructions. It has one live-in register *r5* and one live-out register *r4*. Notice that *r5* is defined locally as well by instruction *I2*. Similarly, *r4* is defined multiple times (*I0* and *I3*); but the last definition in *I3* is treated as live-out.

3.2. Execution Model

In our architecture, the base processing cores are 2-way out-of-order execution engines. They use register map table to rename architectural registers into physical registers. The registers local to a basic block can remain in the core-local register file. However, the live-in and live-out register values have to be communicated among the cores. So we introduce a global register file that is shared across the cores. This includes a global map table and a global physical register file. The live-in registers have to be read from the global register file, while the live-out registers have to be written into the global register file. Thus live-in and live-out registers have to be renamed using the global map table. The sentinel instructions take care of global register renaming.

Any out-of-order execution core should rename registers in program order as well as commit instructions in program order. As the global (inter basic block) dependencies are encapsulated in the sentinel instructions, we only need to ensure that the sentinel instructions execute and rename global registers in program order. We satisfy this program order requirement by using a global program counter (GPC) that is shared across the cores (see Figure 1). The GPC also comes with a locking mechanism such that only one core can access and update the GPC at any point of time [Culler and Singh 1999]. Initially the GPC points to the sentinel instruction in the very first basic block of the program. Once a core fetches the sentinel instruction and completes global register renaming, it updates the GPC to point to the next basic block (i.e., the next sentinel instruction). If the basic block ends with a branch instruction, the GPC will be updated with the predicted target address generated by the branch predictor in the core. Otherwise, the GPC is incremented by the length of the basic block (the length information is encoded in the sentinel instruction). In other words, the GPC always points to a sentinel instruction. After updating, the core releases the lock on the GPC allowing the next core (or the same core) to lock the GPC register. Now the core starts fetching the instructions from the basic block and then executes them out-of-order whenever their operands are ready (i.e., all the local and global dependencies are resolved).

The in-order commit constraint is to handle speculative execution and precise exception. The in-order commit requirement is handled through a shared ticket lock mechanism. The ticket lock contains two registers: *serving* and *ticket* (see Figure 1). Both are initialized to 0. The *ticket* register is used in order to keep track of the order in which the basic blocks are fetched by the cores. When a core locks the GPC, it also reads and increments the current value of *ticket* register. It then tags the reorder buffer

Fig. 5. Bahurupi distributed execution model: (a) Control flow graph (CFG) of a program, and (b) Execution of the CFG on 2-core Bahurupi architecture.

(ROB) entries of all the instructions in the basic block with this ticket value. That is, each basic block of instructions is tagged with a unique ticket value and the ticket values are assigned to basic blocks in program order.

The *serving* register dictates which set of instructions are allowed to be committed. At any point in time only one core is permitted to commit instructions. That is the core for which the instructions are ready to be committed and their associated ticket number matches the value held by the *serving* register. The *serving* register is incremented after all the instructions from the basic block are committed. This process ensures that the basic blocks are committed in order.

Example. Figure 5 illustrates Bahurupi execution model with an example. The left hand side of the figure shows a simple control flow graph (CFG) corresponding to a program. This CFG contains five basic blocks B0–B4. In the beginning, the global program counter (GPC) points to the sentinel instruction of B0. Let us assume that core 0 manages to get a lock on GPC first. It fetches the sentinel and renames the global registers according to live-in and live-out information. The sentinel also indicates that basic block B0 ends in a branch instruction. Therefore core 0 performs branch prediction, which indicates B1 as the predicted next basic block. So GPC is updated to point to the sentinel of B1 and core 0 releases its lock on GPC. During this period core 1 is sitting idle as it cannot obtain a lock on GPC.

Now core 0 starts fetching, decoding, renaming, and executing the regular instructions from basic block B0. Meanwhile, as GPC has been released, core 1 locks GPC and renames global registers corresponding to basic block B1. As B1 does not have any branch instruction in the end, GPC is incremented by the length of B1 and now points to B3. Core 1 also releases the lock on GPC. At this point, both core 0 and core 1 are fetching, renaming, and executing instructions from their corresponding basic blocks.

When core 1 releases the lock, both the cores are still busy fetching instructions from their respective basic blocks. Therefore, none of them attempt to lock the GPC. Only when a core completes fetching all the instructions from its current basic block, it will proceed to get the lock for the next basic block. This is the reason why there is a gap between the completion of execution of sentinel in B1 and the fetching of the sentinel in B3. Next, core 0 completes fetching all the instructions of B0 and locks the GPC for B3. So when core 1 completes fetching all its instructions from basic block B1, it needs to wait for the lock. Even though in this example the basic blocks alternate between

the two cores, it is possible for a core to fetch consecutive basic blocks specially when the other core is handling a large basic block and cannot request lock on GPC.

Now let us focus on the commit. Initially the value of *serving* and *ticket* are both 0. Thus core 0 tags all the instructions of B0 with 0 and *ticket* is incremented to 1. Once the first instruction of B0 completes execution, core 0 can start committing instructions as *serving* value matches the tag of its instructions. Core 1 has tagged all the instructions of B1 with ticket value 1. So core 1 cannot commit in parallel with core 0. Instead, it should wait till core 0 completes all the commit and increments *serving* to 1. This introduces idle cycles in the commit stage of core 1 as shown in Figure 5.

In summary, Bahurupi execution model requires (a) in-order fetching of the sentinels so that global register renaming can happen in-order, and (b) in-order commit of the instructions across the cores. The dashed lines in Figure 5 highlight the in-order fetch and commit. This can introduce idle cycles but is necessary to maintain correctness of program execution. The fetch, rename, and execute of regular instructions in the cores can proceed in parallel to create the illusion of a single virtual ooo engine.

Bahurupi model of execution can even outperform true out-of-order execution engines. For example, for floating point benchmarks, 2-core Bahurupi architecture performs better than 4-way issue out-of-order processor (see Figure 7). This is because Bahurupi can look far ahead in the future. In Figure 5, for example, the cores are fetching instructions from basic blocks B3 and B4 in parallel. In the 4-way issue processor, however, the instructions from B3 and B4 have to be fetched sequentially. As dependencies between the basic blocks are resolved with the help of live-in, live-out information, Bahurupi can exploit ILP across basic blocks much more easily.

4. ARCHITECTURAL DETAILS

Bahurupi architecture uses classic register renaming for both local and global register files. As mentioned before, we introduce a *shared global register file* that includes *global register map* and *global physical register file*. The size of the global register map is determined by the number of registers in the processor ISA. The size of the global physical register file, however, depends on the fraction of register accesses that require global register file. In our architecture, we allocate 40 entries for the global physical register file based on empirical evaluation.

4.1. Live-In Register Renaming

When a core fetches a sentinel instruction, it has to handle the live-in and live-out registers. For a live-in register, there can be two scenarios. In the first case, the value corresponding to the global register has already been produced by a previous basic block. The core only needs to copy this value into the local register file. In the second case, the value is not yet ready. The register renaming logic then simply copies the mapping from the global register map to the local register map. That is, the local register map for the live-in register now points to a global register file entry.

Figure 6 shows an example of register renaming with a basic block. The basic block has one live-in register (r5) and one live-out register (r4). When the sentinel instruction is fetched, the core accesses the global register map where r5 has been renamed to global physical register GR3. Therefore, in the local register map as well we map r5 to GR3. When the regular instructions are fetched within the basic block, they get renamed as usual using *only* the local register map. Hence, source register r5 in instructions I0, I1, I2 get renamed to GR3. Instruction I2 however redefines register r5. At this point, r5 gets renamed to a local physical register LR9. So the next instruction I3 uses LR9 for source register r5. So the same register r5 initially gets mapped to a global physical register and then gets mapped to a local physical register. On the other hand, register

Basic block	Renamed registers	Live-out map
sen 5 0 r5 0 0 r4 0 0		4 GR8 -
I0: addiu r4 r5 -27	I0: addiu LR7 GR3 -27	4 GR8 ROB6
I1: addu r6 r5 r4	I1: addu LR8 GR3 LR7	4 GR8 ROB6
I2: addiu r5 r5 2	I2: addiu LR9 GR3 2	4 GR8 ROB6
I3: addu r4 r5 r4	I3: addu LR10/GR8 LR9 LR7	4 GR8 ROB9
I4: sw r6 (r4)	I4: sw LR8 (LR10)	4 GR8 ROB9

Fig. 6. Global and local register renaming.

r6 in instruction I1 is always mapped to local physical register as it does not belong to live-in register list.

4.2. Live-Out Register Renaming

The core needs to rename the live-out registers. This process is a bit more involved. First the core requests the global register renaming logic to supply a free global physical register corresponding to each live-out register. This mapping information is maintained in the global register map as well as in a special local table called *live-out map*. The live-out map contains only three entries corresponding to three live-out registers. Each entry is a 3-tuple containing (a) architectural register index, (b) global physical register index, and (c) ROB ID of the instruction that last mapped the corresponding architectural register. Figure 6 shows the live-out map of register r4 to free global physical register GR8. The ROB entry is not yet known.

Note that we do not immediately copy the live-out mapping into the local register map. This is because a live-out register can be defined multiple times within a basic block and only the last write to the register should be communicated to the global register file. For example, in Figure 6, live-out register r4 gets defined in both instruction I0 and I3. However, only I3 should write to the global register file.

In a 2-way out-of-order processor, we need to rename 4 source registers and 2 destination registers per cycle. In contrast, sentinel instruction requires renaming 3 source registers and 3 destination registers. However, unlike normal instructions where the hardware needs to identify possible dependencies among the registers being renamed in parallel, we only need to identify 3 free physical registers for sentinel instructions. Thus it is easy to rename 3 registers in one clock cycle for sentinel instruction.

The fetching and renaming of regular instructions in the basic block proceeds as usual. For example, r4 gets renamed to local physical register LR7 in I0 and then to LR10 in I3. Whenever r4 gets renamed, the ROB ID of the corresponding instruction is copied into the live-out map as shown in Figure 6. Originally, the ROB ID is ROB6 corresponding to I0 and then it changes to ROB9 corresponding to I3.

Whenever a normal instruction with live-out register gets renamed, we copy the global register file index into the corresponding ROB entry. Later on, if the live-out register is renamed again in the basic block, the global register file index is removed from the previous ROB entry and added to the new ROB entry. For example, initially ROB entry ROB6 for instruction I0 contains global register index GR8 and is later removed when ROB9 for I3 is updated with global register file index GR8.

When an instruction that defines a live-out register completes execution, it writes the value into both the local and global register file. For example, when I3 completes execution, the value of register r4 will be written to both LR10 and GR8. In addition,

when a core writes to the global physical register file, it needs to broadcast the information on the coalition bus so that other cores dependent on this register can get the value. Finally, when such an instruction is ready to commit, the value is committed to global register file.

It is possible that a regular instruction that last defines a live-out register value (e.g., I3) completes execution even before all the instructions corresponding to its basic block have been fetched and renamed. In that case, when the instruction executed, it was not guaranteed that it needs to write the value into the global register file. This information is known only when all the instructions in the basic block have been renamed. For this reason, instructions from a basic block are not allowed to broadcast and commit live-out registers till all the instructions from that basic block have been fetched and renamed.

As mentioned before, we can only allow in-order commit to support speculation. Even though the cores in a coalition can perform fetch, rename, execute, and register writes in parallel, only one core can perform commit per cycle. However, we are not restricted to at most 2 instructions commit per cycle. This is because all the instructions with local register destination do not need to commit. Only the instructions with live-out destination registers and memory instructions need to commit to the global register file. So we are only restricted to commit at most 2 instructions with live-out destinations and memory instructions per cycle. Hence when we attempt to emulate 4-way or 8-way processor, the commit stage does not become the bottleneck.

4.3. Branch Misprediction and Exceptions

When a core detects a branch misprediction, it will signal all the cores (including itself) to flush the fetch queues and the ROB entries with *ticket* value greater than the *ticket* value of the mispredicted branch instruction. In other words, all the instructions subsequent to the mispredicted branch are flushed from the pipeline similar to what happens in a normal out-of-order execution. The core will ask any other core locking the GPC to release it. The core with the mispredicted branch will then lock the GPC, restore the global *ticket* value to one plus the *ticket* value of the mispredicted branch instruction and set the GPC to the correct address. Then it will continue fetching from the new GPC address which now points to the correct sentinel instruction. The same policy is followed to maintain precise exception.

4.4. Memory Hierarchy

Figure 1 depicts an example of a Bahurupi processor with eight 2-way cores where at most four cores can be composed together. To form a coalition, the cores need to share the data and instruction caches. We employ reconfigurable L1 instruction cache and L1 data cache for this purpose [Chiou et al. 1999]. Both L1 instruction and data cache have four banks. In normal mode, each core is allocated a cache bank, which behaves as a direct-mapped cache and the cache mapping is configured accordingly. In coalition mode, all the four cache banks together behave as a large 4-way set-associative shared cache. The combined instruction and data L2 cache is shared across all the cores both in normal mode and coalition mode.

4.5. Memory Hazards

As with any out-of-order architecture, our design restricts the store operations to update the memory in the commit stage. Thus we can avoid write-after-write memory hazards. However, we still have the problem of write-after-read and read-after-write memory hazards. A load and store executing on different cores can access the same memory location. In that case, we have to ensure that they execute in the original program order. If a load instruction is about to read from an address at which a previous

store (from another core) has to write, then the load operation may read a wrong value. However, this problem is not unique to Bahurupi architecture. Even in a single-core traditional out-of-order execution engine, a load may execute while the memory address of a previous store instruction is still not ready. Later on, it may turn out that the load and the previous store access the same address and hence the value read by the load operation is incorrect. This is handled through memory disambiguation at commit stage by out-of-order processors. We use the same mechanism in Bahurupi. The mechanism enforces that all load operations should execute two times. First, when their operands are ready (execution stage) and second, when they are ready to commit. When the load is executed the second time, it will check if the value read is different from the value which was obtained at the first attempt. If it is different, then it means that another core has committed previously and it wrote at the same address. All the instructions executed after the load instruction by all the cores (including the current one) are corrupted and the core will have to signal all the cores to flush their internal structures and again prepare the GPC and *ticket* registers for a new fetch.

4.6. Reconfiguration Overhead

The main advantage of Bahurupi is that it is a reconfigurable heterogeneous architecture. At runtime, it is possible to form coalition of two to four cores if we need to execute high ILP applications. On the other hand, the architecture behaves like a traditional homogeneous multi-core architecture in non-coalition mode. The reconfiguration overhead of Bahurupi is minimal except for the latency to flush the pipeline. A special instruction is used to request a core to join a coalition or leave from the coalition. In coalition mode, the additional coalition logic is simply turned off. When forming or leaving coalition, the L1 cache memories have to be reconfigured to either run in partitioned mode (for individual cores) or shared mode (for coalition cores). We assume 100 cycle latency for reconfiguration.

4.7. Compiler Support

Any optimizing compiler computes the live-in and live-out information for a basic block. We simply use this information. We modify the compiler to insert the sentinel instruction at the beginning of each basic block as well as split a basic block if it exceeds the threshold for either number of instructions or number of live-in/live-out registers.

5. EXPERIMENTAL EVALUATION

In this section we present an experimental evaluation of Bahurupi architecture. First, we estimate the area and delay overhead due to the coalition logic through synthesis. This is followed by performance and energy evaluation with cycle-accurate architectural simulator. We first present detailed performance evaluation results of individual sequential code to show the effectiveness of the coalition. This is followed by a study of multiprogrammed workload to show the benefit of a reconfigurable heterogeneous architecture like Bahurupi.

5.1. Area and Delay Overhead

In order to estimate the area and delay overhead due to the coalition logic, we synthesize a preliminary version of the baseline processor core and the coalition logic. We use a synthesizable version of the SimpleScalar processor core [Choudhary et al. 2011]. We generate a 2-way and a 4-way SimpleScalar core with the same configuration parameters given in Table II. The synthesis is performed using Synopsys 2010 [Synopsys 2010] design compiler with FreePDK [FreePDK 2011] 45nm technology library.

The main components of the shared resources are represented by the global register file and the corresponding renaming logic that we implement and synthesize. The

renaming logic maps 3 live-in and 3 live-out registers per cycle. Note that normal 2-way processor core is expected to map 4 live-in and 2 live-out registers per cycle whereas a normal 4-way processor has to map 8 live-in and 4 live-out registers per cycle. Further, we assume 3 register read ports and 2 register write ports for the global register file. This is because the global register file is read during the renaming of global live-in registers and there are at most 3 live-in registers per sentinel instruction. The global register file is written when an instruction with live-out destination register completes execution. As we will show later, only 24% of the registers are renamed to global register file for embedded benchmarks. Even for a 4-core coalition, this corresponds to $2\text{-way} \times 4\text{cores} \times 0.24 = 2$ global destination registers written per clock cycle.

Synthesis results show that the baseline 2-way SimpleScalar core can run at 0.6ns clock period. Note that both the synthesizable version of SimpleScalar core we are using Choudhary et al. [2011] as well as the global register file we have designed are not highly optimized. The 4-way baseline SimpleScalar core can run at 0.8ns clock period. In contrast, the global register file can still be synthesized at well below 0.6ns clock period. In other words, the global register file does not contribute to additional delay in the clock period of the baseline 2-way processor core. Thus Bahurupi architecture with coalition logic can easily run at 0.6ns clock period as opposed to 0.8ns clock period for 4-way baseline superscalar processor core.

In terms of area overhead, the coalition logic corresponds to 31.27% of the area of a 2-way SimpleScalar core (the area of the core does not include the caches and the TLB) and 23.48% of the area of a 4-way SimpleScalar core.

5.2. Cycle-Accurate Simulation Setup

5.2.1. Simulator. We implement a cycle-accurate simulator for Bahurupi architecture by modifying an existing multi-core version [Baldawa 2007] of SimpleScalar simulator [Austin et al. 2002]. Our simulated architecture comprises of 2-way out-of-order SimpleScalar cores that share the same memory hierarchy. We simulate 2-core and 4-core Bahurupi architecture built from two 2-way and four 2-way basic cores, respectively. Our Bahurupi cycle-accurate simulator can directly execute sequential program binary annotated with sentinel instructions generated by our modified gcc compiler. We verify that the output produced by Bahurupi simulator matches with the original program output confirming its functional correctness.

For comparison purposes, we also simulate baseline 2-way, 4-way, and 8-way processor architectures. Table II summarizes the processor configuration parameters. The parameters generally get doubled as processor complexity increases. We use 1 cycle latency for L1 data and instruction cache, 6 cycles for L2 data and instruction cache and 18 cycles for the main memory. As mentioned before, our synthesis results indicate that both baseline 2-way processor core and Bahurupi architecture with coalition logic can run at the same clock frequency (0.6ns clock period). The 4-way baseline processor clock can be synthesized at only 0.8ns clock period. However, in the simulation, we optimistically assume that all the baseline architectures (2-way, 4-way, and 8-way) can run at the same clock frequency.

5.2.2. Compiler. We use gcc-2.7.2.3 compiler configured for SimpleScalar PISA instruction set. We modify the compiler to identify basic blocks and live-in, live-out information. The compiler adds the necessary sentinel instructions in the binary executable to support Bahurupi execution model. Moreover, the compiler splits a basic block into two or more blocks when either the size restriction of 16 instructions or 3 live-in, 3 live-out registers restrictions are violated.

5.2.3. Benchmarks. We simulate serial benchmarks from both general-purpose and embedded domain exhibiting a diverse workload. Table I depicts the summary of the

Table I. Workloads Used for Simulation

Workload	Version	Type	Input arg./cycles
go	SPEC95	INT	50 9 2stone9.in/470M
compress	SPEC95	INT	10000 q 2131/27M
li	SPEC95	INT	2 test.lsp/832M
jpeg	SPEC95	INT	specmun.ppm quality 40/49M
perl	SPEC95	INT	primes.pl/8M
fpppp	SPEC95	FP	natoms.in/7M
wave5	SPEC95	FP	wave5.in/575M
parser	SPEC2000	INT	2.1.dict first phrase/180M
bzip2	SPEC2000	INT	input.program/1.71B
gzip	SPEC2000	INT	input.compressed/1.58B
mesa	SPEC2000	FP	test meshfile mesa.in/179M
equake	SPEC2000	FP	test inp.in/808M
swim	SPEC2000	FP	test swim.in/150M
mgrid	SPEC2000	FP	test mgrid.in/259M
applu	SPEC2000	FP	test applu.in/528M
hammer	SPEC2006	INT	num 200 bombesin.hmm/192M
lbm	SPEC2006	FP	10 out 0 1 100.100.130_cf.a.of/29M
crc	MiBench	telecomm	small.pcm/28M
dijkstra	MiBench	network	small input.dat/57M
rijndael (encrypt)	MiBench	security	input.saml.asc/36M
rijndael (decrypt)	MiBench	security	output.small.enc/36M
susan (edges)	MiBench	automotive	input.small.pgm/3M
susan (corners)	MiBench	automotive	input.small.pgm/1M
susan (smoothing)	MiBench	automotive	input.small.pgm/50M
MPEG2 (decode)	Mediabench2	media processing	input.base.4CIF_96bps.mpg/1.72B
MPEG2 (encode)	Mediabench2	media processing	input.base.4CIF_96bps.15.par/534M

Table II. Parameters for Baseline Cores

Parameter	2-way	4-way	8-way
ROB Size	64	128	256
Int ALU	2	4	8
Int MULT	1	2	4
FP ALU	2	4	8
FP MULT	1	2	4
Predictor	2K Comb	4K Comb	8K Comb
L1-D\$	256KB	512KB	1MB
L1-I\$	256KB	512KB	1MB
L2\$	1MB	2MB	4MB

selected workloads. The *cycles* column shows the number of cycles to execute the benchmarks on a 2-way core. As the gcc compiler supported by SimpleScalar PISA instruction set is rather old, we had difficulty in compiling the remaining benchmarks from the suites. Note that this is inherent problem with older version of gcc and has nothing to do with our compiler modifications.

5.3. Sequential Workload

In this section, we present detailed performance analysis of Bahurupi architecture with individual sequential code in comparison with baseline superscalar cores.

5.3.1. Overall Speedup. Figure 7 shows Bahurupi's speedup compared to baseline out-of-order architectures. The speedup is normalized with respect to the performance of a 2-way architecture. As expected, baseline 4-way and 8-way architectures perform much better than 2-way architecture due to more aggressive ILP exploitation—an average speedup of 2.6 for 4-way and 3.9 for 8-way in embedded benchmarks. Note that the speedup numbers are quite optimistic for 4-way and 8-way cores as we assume the

Fig. 7. Bahurupi speedup normalized to 2-way core for (a) SPEC and (b) embedded benchmarks.

same clock frequency and pipeline for all the baseline cores. As our synthesis results showed, we would need slower clock or more pipeline stages for 4-way and 8-way cores.

What is interesting here is that Bahurupi matches and sometimes even exceeds the performance of true out-of-order execution engines. On an average, for embedded benchmarks, dual-core and quad-core Bahurupi outperforms 4-way and 8-way architectures, respectively. The speedup numbers are even more impressive for floating-point SPEC benchmarks, where, on an average, dual-core and quad-core Bahurupi improve performance by 23% and 57% compared to 4-way and 8-way cores. Even for SPEC integer benchmarks, dual-core and quad-core Bahurupi achieves, on an average, 92% and 91% of the performance of 4-way and 8-way cores. In summary, for different workloads, Bahurupi can achieve 1.31–5.61 speedup through core coalition compared to baseline 2-way cores used in normal mode.

The reason that 2-core (4-core) Bahurupi can outperform 4-way (8-way) superscalar architecture is because Bahurupi can exploit far-flung ILP. Theoretically, both 4-core Bahurupi coalition and 8-way baseline normal superscalar core can execute at most 8 instructions per cycle. However, the baseline core is mostly restricted to finding ILP within one or two basic blocks. In contrast, the register dependencies among the basic blocks are identified at compile time and explicitly specified in the binary executable in Bahurupi architecture. This allows the 4 cores in Bahurupi architecture to work on independent instructions from 4 different basic blocks leading to higher ILP. As a concrete example, consider a loop containing only one large basic block with many intra-loop dependencies but no loop carried dependencies. An 8-way superscalar core would fetch one iteration of the loop at a time and would not find many independent instructions to execute due to data dependencies. A 4-core Bahurupi, on the other hand, would fetch and execute 4 independent iterations of the loop in parallel and thus would be able to discover more ILP.

Fig. 8. Bahurupi energy consumption normalized to 2-way core for (a) SPEC and (b) embedded benchmarks.

5.3.2. Energy Consumption. Next we evaluate the energy consumption of Bahurupi architecture compared to regular out-of-order cores. We use SimWattch [Brooks et al. 2000] suitably modified to compute the energy consumption in the cores. We use CACTI [Shivakumar et al. 2001] to model power consumption of the global register file, global renaming logic, and the broadcast bus and add this energy consumption in the coalition logic to the overall energy consumption of Bahurupi. Figure 8 plots energy consumption of Bahurupi normalized with respect to the baseline 2-way core. As expected, for SPEC integer benchmarks, 2-core Bahurupi consumes 5% more energy compared to 4-way baseline cores due to slightly increased execution time and the extra energy consumption due to coalition logic and sentinel instructions. But 4-core Bahurupi improves energy consumption by 29% compared to 8-way baseline architecture. For SPEC floating point and embedded benchmarks, on an average, 2-core Bahurupi improves the energy consumption by 26% and 11%, respectively, compared to 4-way cores. This is due to the reduced power per core and the overall improved execution time.

5.3.3. Load Balance. We do not impose any constraint on the order in which the cores can lock the GPC. Instead, we allow a core to lock the GPC when it is free. To evaluate the impact of this design decision on load balancing, we plot the percentage of instructions committed by each core on a 4-core Bahurupi architecture in Figure 9. The figure shows Bahurupi achieves almost perfect load balance among the cores.

5.3.4. Global Register File Access. Bahurupi achieves considerable speedup when register dependencies are mostly restricted within the basic block, i.e., most register accesses are to the local register file. Figure 10 quantizes this characteristics of the benchmarks by plotting the percentage of destination registers that get renamed to global register file compared to the total number of destination registers. On an average, for integer SPEC benchmarks 41% of registers are renamed to global register file, whereas only 27% and 24% of registers are renamed to global register file for SPEC floating point and embedded benchmarks. This contributes to better speedup for SPEC floating point and embedded benchmarks in coalition mode compared to SPEC integer benchmarks.

Fig. 9. Load balance on 4-core Bahurupi.

Fig. 10. Percentage of destination registers renamed to global register file.

Fig. 11. Broadcasts on coalition bus in (a) 2-core Bahurupi and (b) 4-core Bahurupi.

5.3.5. Traffic on Coalition Bus. If multiple cores attempt to use the coalition bus in parallel to broadcast global register file writes, then we have to serialize the writes leading to a bottleneck. Figure 11 shows the average number of broadcasts required on the coalition bus per cycle in 2-core Bahurupi architecture is well below 1.0, while for 4-core Bahurupi, it is below 1.6. This indicates that broadcast due to global register file writes is not a bottleneck. In both cases, floating point applications show higher traffic on the coalition bus as they expose higher ILP.

5.3.6. Sentinel Instruction Overhead. Bahurupi pays the price for reduced hardware complexity with increased code size due to the addition of sentinel instructions. Figure 12 shows that, on an average, code size for SPEC integer applications increases by 24% and the floating point ones by 15%. The Mediabench and MiBench applications size increase by 19% on an average. Note that we did not optimize the code. The code size

Table III. Multi-Core Configurations

Configuration	Description
8x2-way	CMP with eight 2-way cores
4x4-way	CMP with four 4-way cores
2x8-way	CMP with two 8-way cores
1x8-way+4x2-way	Heterogeneous CMP with one 8-way and four 2-way cores
2x4-way+4x2-way	Heterogeneous CMP with two 4-way and four 2-way cores
8-core Bahurupi	Bahurupi Reconfigurable CMP

Fig. 12. Code size increase due to sentinel instructions.

increase can be countered through compiler optimizations such as loop unrolling and superblock formation that increase basic block size.

5.4. Multi-Programmed Workload

We have now established that Bahurupi architecture through the coalition of simple 2-way cores can perform equal to and sometimes better than 4-way or 8-way superscalar cores. However, the true advantage of Bahurupi is that it is a reconfigurable heterogeneous architecture that can be configured at runtime to best match the workload. To emulate the real work scenario where multiple applications are running on multi-core architecture simultaneously, we create multi-programmed workload consisting of sets of 8 benchmarks from Table I. We create workloads with only high ILP benchmarks, low ILP benchmarks, and mixed ILP benchmarks.

We evaluate multiple static homogeneous and heterogeneous multi-core systems as shown in Table III. The first five configurations are fixed and imply a variation in both number of cores and issue width of the cores. CMP (chip multiprocessor) with eight 2-way cores corresponds to a homogenous multi-core system. Similarly four 4-way and two 8-way cores correspond to homogeneous multi-core systems with more complex cores. We have two static heterogeneous multi-core architectures: one complex 8-way core plus four simple cores, and two 4-way cores plus four simple cores. The 8-core Bahurupi architecture is the one shown in Figure 1. The difference between Bahurupi and static heterogeneous architecture is that Bahurupi can be reconfigured at runtime to create many different heterogeneous architectures. The only restriction is that we can have at most two coalitions at any point in time.

For each benchmark, we statically identify the best configuration (2-way, 4-way or 2-core coalition, 8-way or 4-core coalition) to run that benchmark. We develop an oracle scheduler that finds the best scheduling and mapping to minimize the execution time of the given workload (consisting of 8 benchmarks). For Bahurupi architecture, in addition, we select the best configuration for the workload and if necessary perform dynamic reconfiguration. The design of a runtime design space exploration algorithm and scheduler is outside the scope of this paper and is left as future work.

Figures 13 show the results from running sets of eight benchmarks on the given architectures. The Y-axis represents the speedup with respect to the execution time of a workload on eight 2-way cores. The X-axis shows the constitution of the workload.

Fig. 13. Comparison of Bahurupi with static homogeneous and heterogeneous architectures for multi-programmed workload with 8 benchmarks per set.

The execution time of a workload critically depends on the ILP of the applications, their execution time variation, as well as the number of cores and their configurations. For workloads with 8 applications, if the execution time of all the applications were the same, then the 8-core configuration will always perform the best. However, there is a lot of variation in execution time among the applications as shown in Table I. In that case, more complex cores can speed up the execution of long-running high-ILP application while simpler cores can run multiple short-running low-ILP applications during the lifetime of the long-running application. This leads to overall improved execution time. Hence, in general we see that configurations with one or more complex cores perform much better than the baseline homogeneous configuration with only simple cores. Bahurupi outperforms all the static architectures for almost all workloads. Bahurupi uses its flexibility in deciding on the number of cores and their complexity at runtime to best match the workload leading to superior performance.

6. CONCLUSIONS

We have presented a reconfigurable multi-core architecture, called Bahurupi, that can dynamically adapt itself to support both multi-threaded code with explicit thread-level parallelism as well as sequential code with instruction-level parallelism. Bahurupi can dynamically merge the base 2-way out-of-order execution engines to achieve the performance of 4-way or even 8-way out-of-order processors. Bahurupi is a hardware-software cooperative solution that requires minimal additional hardware resources and compiler support for coalition. We are currently working on synthesizing a complete prototype of Bahurupi architecture both in ASIC and FPGA. In the future, we plan to develop design space exploration techniques to select optimal configurations of Bahurupi architectures given a set of applications.

ACKNOWLEDGMENTS

We thank Kaushik Triyambaka Mysur for his help in implementing synthesizable version of the architecture and the anonymous reviewers for their constructive comments that helped improve the paper.

REFERENCES

- AUSTIN, T., LARSON, E., AND ERNST, D. 2002. Simplescalar: An infrastructure for computer system modeling. *Computer* 35, 59–67.

- BALAKRISHNAN, S., RAJWAR, R., UPTON, M., AND LAI, K. 2005. The impact of performance asymmetry in emerging multicore architectures. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*. 506–517.
- BALDAWA, S. 2007. CMPSIM: A flexible multiprocessor simulation environment. M.S. Dissertation, The University of Texas at Dallas.
- BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 72–81.
- BROOKS, D., TIWARI, V., AND MARTONOSI, M. 2000. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*. 83–94.
- CHIOU, D., DEVADAS, S., RUDOLPH, L., AND ANG, B. 1999. Dynamic cache partitioning via columnization. Tech. rep., MIT.
- CHOUDHARY, N. K., WADHAVKAR, S. V., SHAH, T. A., MAYUKH, H., GANDHI, J., DWIEL, B. H., NAVADA, S., NAJAF-ABADI, H. H., AND ROTENBERG, E. 2011. FabScalar: Composing synthesizable RTL designs of arbitrary cores within a canonical superscalar template. In *Proceedings of the 38th IEEE/ACM International Symposium on Computer Architecture (ISCA)*. 11–22.
- CULLER, D. E. AND SINGH, J. P. 1999. *Parallel Computer Architecture: A Hardware/Software Approach*. Elsevier Morgan Kaufmann. p. 337.
- ETSION, Y., CABARCAS, F., RICO, A., RAMIREZ, A., BADIA, R. M., AYGADE, E., LABARTA, J., AND VALERO, M. 2010. Task superscalar: An out-of-order task pipeline. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 89–100.
- FREEPDK 2011. FreePDK. <http://www.eda.ncsu.edu/wiki/FreePDK>.
- HILL, M. D. AND MARTY, M. R. 2008. Amdahl's Law in the multicore era. *Computer* 41, 7, 33–38.
- IPEK, E., KIRMAN, M., KIRMAN, N., AND MARTINEZ, J. F. 2007. Core fusion: Accommodating software diversity in chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*. 186–197.
- JERRAYA, A. AND WOLF, W. 2005. *Multiprocessor Systems-on-Chip*. Elsevier Morgan Kaufmann.
- KIM, C., SETHUMADHAVAN, S., GOVINDAN, M. S., RANGANATHAN, N., GULATI, D., BURGER, D., AND KECKLER, S. W. 2007. Composable lightweight processors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 381–394.
- KUMAR, R., JOUPPI, N. P., AND TULLSEN, D. M. 2004a. Conjoined-core chip multiprocessing. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 195–206.
- KUMAR, R., TULLSEN, D. M., RANGANATHAN, P., JOUPPI, N. P., AND FARKAS, K. I. 2004b. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*. 64–75.
- PATEL, A., AFAM, F., CHEN, S., AND GHOSE, K. 2011. MARSS: A full system simulator for multicore x86 CPUs. In *Proceedings of the 48th Design Automation Conference (DAC)*. 1050–1055.
- SALVERDA, P. AND ZILLES, C. 2008. Fundamental performance constraints in horizontal fusion of in-order cores. In *Proceedings of the 14th International Symposium on High Performance Computer Architecture (HPCA)*. 252–263.
- SANKARALINGAM, K., NAGARAJAN, R., LIU, H., KIM, C., HUH, J., BURGER, D., KECKLER, S. W., AND MOORE, C. R. 2003. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*. 422–433.
- SHIVAKUMAR, P., JOUPPI, N. P., AND SHIVAKUMAR, P. 2001. CACTI 3.0: An integrated cache timing, power, and area model. Tech. rep., HP Labs.
- SOHI, G. S., BREACH, S. E., AND VIJAYKUMAR, T. N. 1995. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*. 414–425.
- SULEMAN, M. A., MUTLU, O., QURESHI, M. K., AND PATT, Y. N. 2009. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 253–264.
- SYNOPSIS 2010. Synopsys. <http://www.synopsys.com>.
- TARJAN, D., BOYER, M., AND SKADRON, K. 2008. Federation: Repurposing scalar cores for out-of-order instruction issue. In *Proceedings of the 45th Annual Design Automation Conference (DAC)*. 772–775.
- VAJAPAYAM, S., RYCHLIK, B., AND SHEN, J. P. 2008. Dependence-chain processing using trace descriptors having dependency descriptors. Intel Corporation Patent No.: US 7363467 B2.

WALL, D. W. 1991. Limits of instruction-level parallelism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 176–188.

ZHONG, H., LIEBERMAN, S. A., AND MAHLKE, S. A. 2007. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA)*. 25–36.

Received July 2011; revised October 2011; accepted November 2011