Conservation Cores: Reducing the Energy of Mature Computations

Ganesh Venkatesh Jack Sampson Nathan Goulding Saturnino Garcia
Vladyslav Bryksin Jose Lugo-Martinez Steven Swanson Michael Bedford Taylor

Department of Computer Science & Engineering
University of California, San Diego
{gvenkatesh,jsampson,ngouldin,sat,vbryksin,jlugomar,swanson,mbtaylor}@cs.ucsd.edu

Abstract

Growing transistor counts, limited power budgets, and the breakdown of voltage scaling are currently conspiring to create a *utilization wall* that limits the fraction of a chip that can run at full speed at one time. In this regime, specialized, energy-efficient processors can increase parallelism by reducing the per-computation power requirements and allowing more computations to execute under the same power budget. To pursue this goal, this paper introduces *conservation cores*. Conservation cores, or *c-cores*, are specialized processors that focus on reducing energy and energy-delay instead of increasing performance. This focus on energy makes c-cores an excellent match for many applications that would be poor candidates for hardware acceleration (e.g., irregular integer codes). We present a toolchain for automatically synthesizing c-cores from application source code and demonstrate that they can significantly reduce energy and energy-delay for a wide range of applications. The c-cores support patching, a form of targeted reconfigurability, that allows them to adapt to new versions of the software they target. Our results show that conservation cores can reduce energy consumption by up to 16.0× for functions and by up to 2.1× for whole applications, while patching can extend the useful lifetime of individual c-cores to match that of conventional processors.

Categories and Subject Descriptors C.1.3 [*Processor Architectures*]: Heterogeneous (hybrid) systems; C.3 [*Special-Purpose and Application-based systems*]

General Terms Design, Experimentation, Measurement

Keywords Conservation Core, Utilization Wall, Heterogeneous Many-Core, Patching

1. Introduction

As power concerns continue to shape the landscape of general-purpose computing, heterogeneous and specialized hardware has emerged as a recurrent theme in approaches that attack the power wall. In the current regime, transistor densities and speeds continue to increase with Moore's Law, but limits on threshold voltage scaling have stopped the downward scaling of per-transistor switching power. Consequently, the rate at which we can switch transistors is far outpacing our ability to dissipate the heat created by those transistors.

The result is a technology-imposed *utilization wall* that limits the fraction of the chip we can use at full speed at one time. Our experiments with a 45 nm TSMC process show that we can switch less than 7% of a 300mm^2 die at full frequency within an 80W power budget. ITRS roadmap projections and CMOS scaling theory suggests that this percentage will decrease to less than 3.5% in 32 nm, and will continue to decrease by almost half with each process generation—and even further with 3-D integration.

The effects of the utilization wall are already indirectly apparent in modern processors: Intel's Nehalem provides a "turbo mode" that powers off some cores in order to run others at higher speeds. Another strong indication is that even though native transistor switching speeds have continued to double every two process generations, processor frequencies have not increased substantially over the last 5 years.

In this regime, reducing per-operation energy [19] translates directly into increased potential parallelism for the system: If a given computation can be made to consume less power at the same level of performance, other computations can be run in parallel without violating the power budget.

This paper attacks the utilization wall with *conservation cores*. Conservation cores, or *c-cores*, are application-specific hardware circuits created for the purpose of reducing energy consumption on computationally-intensive applications. Since it is no longer possible to run the entire chip at full frequency at once, it makes sense to customize the portions of the chip that are running so they will be as efficient as possible for the application at hand. In effect, conservation cores allow architects to trade area for energy in a processor's design. The utilization wall has made this trade-off favorable, because Moore's Law has made increases in transistor counts cheap, while poor CMOS scaling has exhausted power budgets, making increases in power consumption very expensive.

Conservation cores have a different goal than conventional application-specific circuits, and we differentiate between c-cores and the more common *accelerators* along several axes. First, accelerators focus on improving performance, at a potentially worse, equal, or better energy efficiency. Conservation cores, on the other hand, focus primarily on energy reduction. C-cores that are also accelerators are possible, but this work targets similar levels of performance, and focuses on reducing energy and energy-delay, especially at advanced technology nodes where DVFS is less effective for saving energy.

Shifting the focus from performance to efficiency allows c-cores to target a broader range of applications than accelerators. Accelerators provide the greatest benefit for codes with large amounts

ASPLOS'10, March 13–17, 2010, Pittsburgh, Pennsylvania, USA.

of parallelism and predictable communication patterns, since these codes map naturally onto hardware. Thus, parallelism-intensive regions of code that are hot (i.e., occupy a high percentage of running time) are the best candidates for implementation as accelerators. On the other hand, c-cores are parallelism-agnostic: Hot code with a tight critical path, little parallelism, and/or very poor memory behavior is an excellent candidate for a c-core, as c-cores can reduce the number of transistor toggles required to get through that code. For instance, our results show that c-cores can deliver significant energy savings for irregular, integer applications (e.g., MCF from SPEC 2006) that would be difficult to automatically accelerate with specialized hardware.

Incorporating c-cores into processors, especially at a scale large enough to save power across applications with multiple hot spots, raises a number of challenges:

1. **Determining which c-cores to build** In order to build c-cores, we must be able to identify which pieces of code are the best candidates for conversion into c-cores. The code should account for a significant portion of runtime and energy, and stem from a relatively stable code base.
2. **Automatic synthesis** Designing numerous c-cores by hand is not scalable, so it must be possible to synthesize c-cores automatically and correctly, without significant human intervention.
3. **Programming model** It should not be necessary to rewrite applications to make use of c-cores. The system must utilize them automatically.
4. **Longevity** Conservation cores should remain useful even as the code they are designed to replace evolves.
5. **System integration** Since c-cores should work seamlessly with existing code, the c-core hardware and memory model must be tightly integrated with the rest of system.

This paper describes both a system architecture that incorporates the c-cores and the tools for automatically creating them and compiling for them. The toolchain automatically extracts the key kernels in a given codebase and uses a custom C-to-silicon infrastructure to generate 45 nm implementations of the c-cores. The compiler takes in a description of the c-cores available on a chip, and emits code that can utilize the available c-cores.

Conservation cores also support *patching*, a form of load-time reconfiguration, that allows one c-core to run both past and future versions of the code it was generated from. The patching facilities match the types of changes we found in our study of changes in mature code bases (i.e., those in which large-scale changes are infrequent and most modifications between versions are small). This adaptability is essential if c-cores are to be useful on commodity, general-purpose processors, since these designs' lifetimes span many software upgrades.

To evaluate c-cores we have generated 18 fully placed-and-routed c-cores for multiple versions of five applications. The c-cores improve energy efficiency for their target functions by between 3.3× and 16.0×, and reduce system energy consumption by up to 47% and energy-delay by up to 55% at the full application level. Furthermore, their patching capabilities ensure that, despite the appearance of new software versions, they will continue to deliver significant savings for between 9 and 15 years in most cases.

The rest of this paper is organized as follows: Section 2 describes the utilization wall in more detail. Section 3 provides an overview of c-cores and their life-cycle in a computer system. Sections 4 and 5 describe the internal architecture of a c-core and explain our approach to ensuring c-core longevity using patches. Section 6 details our c-core toolchain and methodology. Section 7 presents our results, and Section 8 discusses related work. Finally, Section 9 concludes.

Param.	Description	Relation	Classical Scaling	Leakage Limited
B	power budget		1	1
A	chip size		1	1
V_t	threshold voltage		$1/S$	1
V_{dd}	supply voltage	$\sim V_t \times 3$	$1/S$	1
t_{ox}	oxide thickness		$1/S$	$1/S$
W, L	transistor dimensions		$1/S$	$1/S$
I_{sat}	saturation current	WV_{dd}/t_{ox}	$1/S$	1
p	device power at full frequency	$I_{sat}V_{dd}$	$1/S^2$	1
C_{gate}	**capacitance**	WL/t_{ox}	$\mathbf{1/S}$	$\mathbf{1/S}$
F	**device frequency**	$\frac{I_{sat}}{C_{gate}V_{dd}}$	$\mathbf{S}$	$\mathbf{S}$
D	**devices per chip**	$A/(WL)$	$\mathbf{S^2}$	$\mathbf{S^2}$
P	**full die, full frequency power**	$D \times p$	$\mathbf{1}$	$\mathbf{S^2}$
U	**utilization at fixed power**	B/P	$\mathbf{1}$	$\mathbf{1/S^2}$

Table 1. The utilization wall The utilization wall is a consequence of CMOS scaling theory and current-day technology constraints, assuming fixed power and chip area. The Classical Scaling column assumes that V_t can be lowered arbitrarily. In the Leakage Limited case, constraints on V_t, necessary to prevent unmanageable leakage currents, hinder scaling, and create the utilization wall.

2. The utilization wall

In this section, we examine the utilization wall in greater detail. First, we show how the utilization wall is a consequence of CMOS scaling theory combined with modern technology constraints. Second, we use the results of our own experiments with TSMC 90 and 45 nm processes to measure the impact of the utilization wall in current process technologies. Finally, we use ITRS predictions for 32 nm to draw conclusions on that process.

Scaling Theory The utilization wall has arisen because of a breakdown of classical CMOS scaling as set down by Dennard [11] in his 1974 paper. The introduction of 3D CMOS technology will exacerbate this trend further.

Table 1 shows how the utilization wall emerges from these trends. The equations in the "Classical Scaling" column governed scaling up until 130 nm, while the "Leakage Limited" equations govern scaling at 90 nm and below. CMOS scaling theory holds that transistor capacitances (and thus switching energy) decrease roughly by a factor of S (where S is the scaling factor, e.g., 1.4×) with each process shrink. At the same time, transistor switching frequency rises by S and the number of transistors on the die increases by S^2.

In the Classical Scaling Regime, it has been possible to scale supply voltage by $1/S$, leading to constant power consumption for a fixed-size chip running at full frequency, and consequently, no utilization wall. Scaling the supply voltage requires that we also scale the threshold voltage proportionally. However, this is not an issue because leakage, although increasing exponentially, is not significant in this regime.

In the Leakage Limited Regime, we can no longer scale the threshold voltage because leakage rises to unacceptable levels.

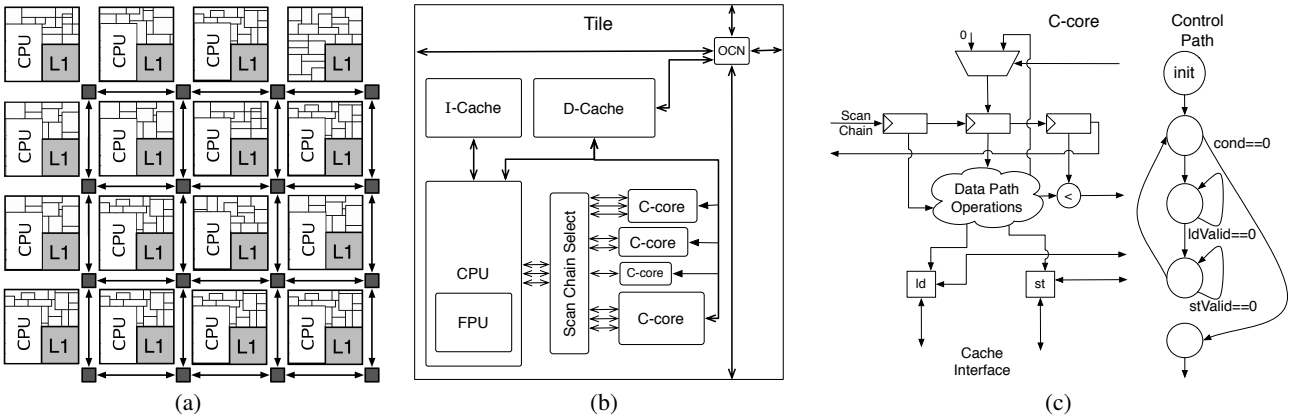

Figure 1. The high-level structure of a c-core-enabled system A c-core-enabled system (a) is made up of multiple individual tiles (b), each of which contains multiple c-cores (c). Conservation cores communicate with the rest of the system through a coherent memory system and a simple scan-chain-based interface. Different tiles may contain different c-cores. Not drawn to scale.

Without the corresponding supply voltage scaling, reduced transistor capacitances are the only remaining counterbalance to increased transistor frequencies and increasing transistor counts. Consequently, the net change in full chip, full frequency power is rising as S^2. This trend, combined with fixed power budgets, indicates that the fraction of a chip that we can run at full speed, or the utilization, is falling as $1/S^2$. Thus, the utilization wall is getting exponentially worse, roughly by a factor of two, with each process generation.

Experimental results To quantify the current impact of the utilization wall, we synthesized, placed, and routed several circuits using the Synopsys Design and IC Compilers. Table 2 summarizes our findings. For each process, we used the corresponding TSMC standard cell libraries to evaluate the power and area of a 300 mm^2 chip filled with 64-bit operators to approximate active logic on a microprocessor die. Each operator is a 64-bit adder with registered inputs and outputs, which runs at its maximum frequency in that process. In a 90 nm TSMC process, running a chip at full frequency would require 455 W, which means that only 17.6% of the chip could be used in an 80 W budget. In a 45 nm TSMC process, a similar design would require 1225 W, resulting in just 6.5% utilization at 80 W, a reduction of 2.6× attributable to the utilization wall. The equations in Table 1 predicted a larger, 4× reduction. The difference is due to process and standard cell tweaks implemented between the 90 nm and 45 nm generations. Table 2 also extrapolates to 32 nm based on ITRS data for 45 and 32 nm processes. Based on ITRS data, for the 32 nm process, 2401 W would be required for a full die at full frequency, resulting in just 3.3% utilization.

Process	90 nm TSMC	45 nm TSMC	32 nm ITRS
Frequency (GHz)	2.1	5.2	7.3
mm^2 Per Op.	.00724	.00164	.00082
# Operators	41k	180k	360k
Full Chip Watts	455	1225	2401
Utilization at 80 W	17.6%	6.5%	3.3%

Table 2. Experiments quantifying the utilization wall Our experiments used Synopsys CAD tools and TSMC standard cell libraries to evaluate the power and utilization of a 300 mm^2 chip filled with 64-bit adders, separated by registers, which is used to approximate active logic in a processor.

Discussion The effects of the utilization wall are already indirectly apparent in modern processors: Intel's Nehalem provides a "turbo mode" that powers off some cores in order to run others at higher speeds. Another strong indication is that even though native transistor switching speeds have continued to double every two process generations, processor frequencies have not increased substantially over the last 5 years. The emergence of three-dimensional (3D) CMOS integration will exacerbate this problem by substantially increasing device count without improving transistor energy efficiency.

For scaling existing multicore processor designs, designers have choices that span a variety of design points, but the best they can do is exploit the factor of S (e.g., 1.4×) reduction in transistor switching energy that each generation brings. Regardless of whether designers a) increase frequency by a factor of 1.4×, b) increase core count by 1.4×, c) increase core count by 2×, and reduce frequency by 1.4×, or d) some compromise of the three, the utilization wall ensures transistor speeds and densities are rapidly out-pacing the available power budget to switch them. Conservation cores are one mechanism for addressing this issue: Specialized silicon can trade area for energy efficiency and enable systems with higher throughput.

3. Conservation cores: System overview

This section provides an overview of c-core-enabled systems. It describes the composition of a prototypical c-core system and the c-core synthesis process. Then, it outlines our approach to compilation and handling target application program changes.

3.1 Basic hardware architecture

A c-core-enabled system includes multiple c-cores embedded in a multi- or many-core tiled array like the one in Figure 1(a). Each tile of the array contains a general purpose processor (the "CPU"), cache and interconnect resources, and a collection of tightly-coupled c-cores. The c-cores target hot regions of specific applications that represent significant fractions of the target system's workload. The CPU serves as a fallback for parts of applications that are not important enough to be supported by the c-cores or that were not available at the time of the manufacture of the chip.

Within a tile (Figure 1(b)), the c-cores are tightly coupled to the host CPU via a direct, multiplexed connection to the L1 cache, and by a collection of scan chains that allow the CPU to read and

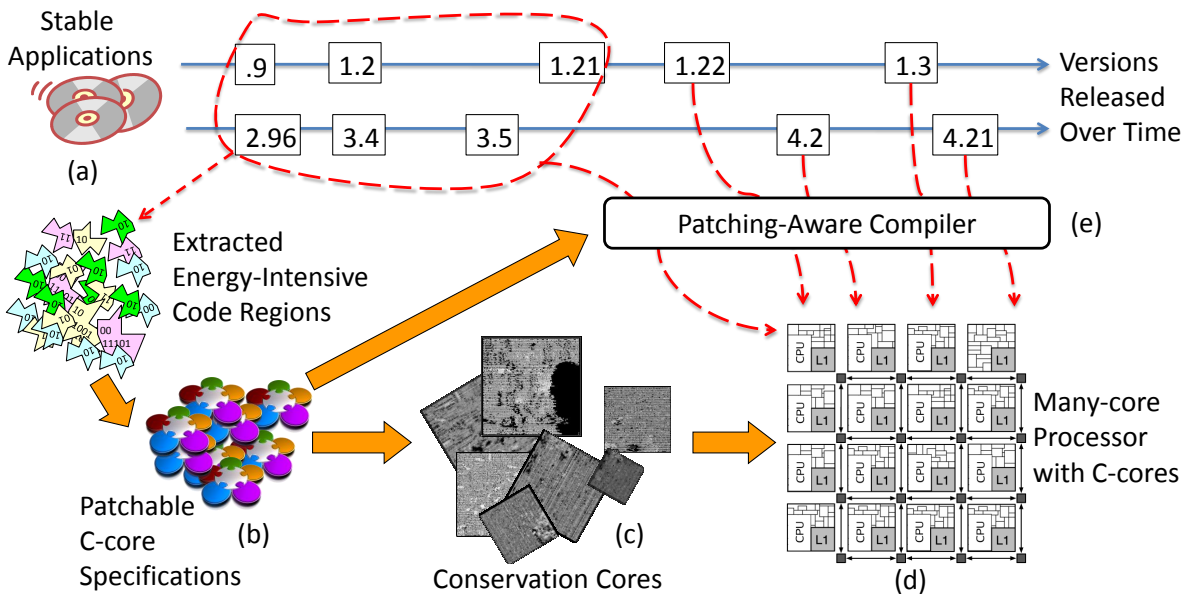

Figure 2. Automatic synthesis and compilation of c-cores A profiler (a) extracts a set of energy-intensive code regions from a corpus of stable applications in the processor's target domain. The toolchain generalizes these regions into patchable c-core specifications (b), translates them into circuits via an automated synthesis infrastructure (c), and integrates them into a multi-core processor (d). A patching-aware compiler maintains a database of c-core specifications and generates binaries that execute on a combination of c-cores and the local CPU. The compiler (e) generates a patching configuration that allows the c-cores to run future (and past) versions of the original applications.

write all state within the c-cores. The CPU uses these scan chains for passing arguments, for context switching, and for patching the c-cores. These facilities allow the system to reconfigure a c-core to run future and past modified versions of the source code that was used to generate the c-cores. Most data transfer occurs through the coherent L1 cache connection.

An individual c-core (Figure 1(c)) comprises a datapath and control state machine derived directly from the code it targets. Specialized load and store units share ports to memory and use a simple token-based protocol to enforce correct memory ordering and support nearly arbitrary C code. The scan chain interface provides access to internal state.

3.2 Creating and targeting conservation cores

Figure 2 depicts the generation of a many-core processor equipped with c-cores and the toolchain needed to target them. The process begins with the processor designer characterizing the workload by identifying codes that make up a significant fraction of the processor's target workload. The toolchain extracts the most frequently used (or "hot") code regions (a), augments them with a small amount of reconfigurability (b) and then synthesizes c-core hardware (see Section 4) using a 45 nm standard cell CAD flow (c). A single processor contains many tiles, each with a general purpose CPU and collection of different c-cores (d).

In order to generate code for the processor, we extend a standard compiler infrastructure (which could easily be a derivative of standard compiler infrastructures such as GCC, the Intel C++ compiler, or Microsoft Visual Studio—in our case, we use a combination of OpenIMPACT and GCC) to support automatic code generation for c-cores. The compiler incorporates a description of the c-core that the manufacturer has shipped silicon for. The compiler uses a matching algorithm to find similarities between the input code and the c-core specifications (e). In cases where there are close matches, the compiler will generate both CPU-only object code and object code that makes use of the c-core. The latter version of the code includes patching information that is downloaded into the c-core via scan chain before it is used. The decision of whether to use the c-core-enabled version of the code or the "software-only" version, is made at run time, based on c-core availability and other factors.

3.3 Support for future application versions

Although the c-cores are created to support existing versions of specific applications, they also need to support newer versions that are released after the original c-cores were synthesized. To do this, the c-cores include reconfiguration bits which allows the behavior of c-cores to adapt to commonly found changes in programs. The patching engine in the compiler performs a specialized graph-matching algorithm on the data- and control-flow graphs of the application or library (see Section 5). Depending on the extent of the differences between the versions, the patch configuration state may specify small changes (e.g., replacing an addition with a subtraction or replacing a constant value) or larger changes, such as bypassing a long sequence of operations with software execution on the CPU. This patch is automatically included in the program binary by the compiler and is invisible to the programmer or user.

The next two sections provide a more complete discussion of each component of the system.

4. Conservation core architecture

This section describes the architecture of a c-core-enabled system in more detail. We describe the organization and synthesis of the c-cores themselves, the interface between the c-cores and the rest of the system, and their support for patchability.

4.1 Conservation core organization

Figure 1(c) shows the architecture of a prototypical c-core. The principle components are the datapath, the control unit, the cache interface, and the scan chain interface to the CPU.

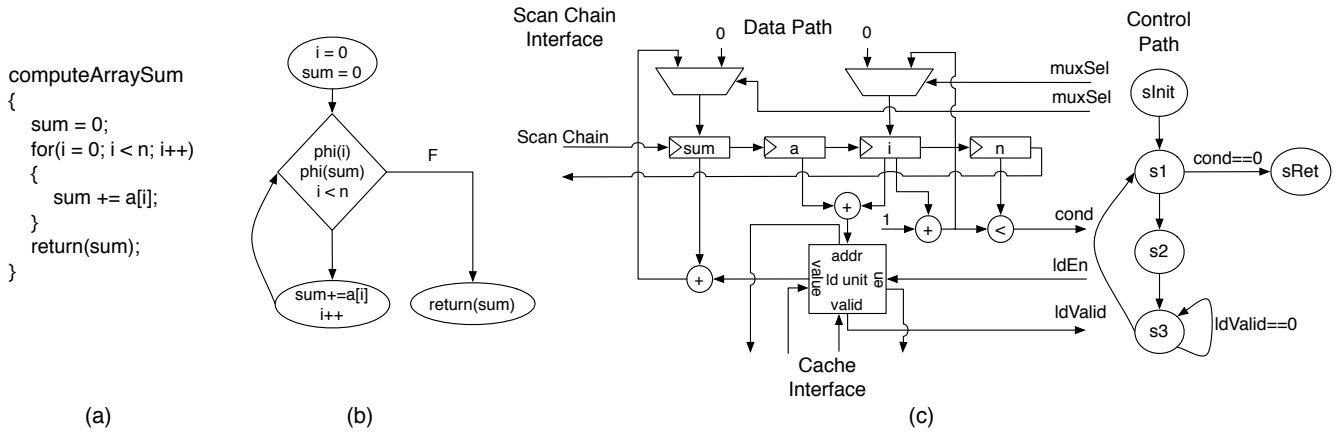

Figure 3. Conservation core example An example showing the translation from C code (a), to the compiler's internal representation (b), and finally to hardware (c). The hardware schematic and state machine correspond very closely to the data and control flow graphs of the C code.

Datapath and control By design, the c-core datapath and control very closely resemble the internal representation that our toolchain extracts from the C source code. The datapath contains the functional units (adders, shifters, etc.), the muxes to implement control decisions, and the registers to hold program values across clock cycles.

The control unit implements a state machine that mimics the control flow graph of the code. It tracks branch outcomes (computed in the datapath) to determine which state to enter on each cycle. The control path sets the enable and select lines on the registers and muxes so that the correct basic block is active each cycle.

The close correspondence between the program's structure and the c-core is important for two reasons: First, it makes it easier to enforce correct memory ordering in irregular programs. The ordering that the control path's state machine enforces corresponds to the order that the program counter provides in general purpose processors, and we use that ordering to enforce memory dependencies. Second, by maintaining a close correspondence between the original program and the hardware, it is more likely that small changes in the source code (which are the common case) will result in correspondingly small patches to the hardware.

To maintain this correspondence and to reduce the number of registers required in the datapath, the registers in the c-core datapaths adhere to SSA form: Each static SSA variable has a corresponding register. This invariant minimizes the number of register updates: Exactly one register value changes per new value that the program generates.

Memory interface and ordering Memory operations require special attention to ensure that the c-core enforces memory ordering constraints. Conservation cores enforce these constraints by allowing only one memory operation per basic block. The c-core only activates one basic block at a time, guaranteeing that memory operations execute in the correct order.

The load/store units connect to a coherent data cache that ensures that all loads and stores are visible to the rest of the system regardless of which addresses the c-core accesses.

Since memory operations can take multiple cycles, the toolchain adds a self-loop to the basic block that contains each memory operation and exports a "valid" line to the control path. When the memory operation is complete, it asserts the "valid" signal and control exits the loop and proceeds with the following basic block. The "valid" signal is similar to the memory ordering token used in systems such as Tartan [27] and WaveScalar [35].

Most of the communication between c-cores and the CPU occurs via the shared L1 cache. A coherent, shared memory interface allows us to construct c-cores for applications with unpredictable access patterns. Conventional accelerators cannot speed up these applications, because they cannot extract enough memory parallelism. Such applications can be an excellent fit for c-cores, however, as performance is not the primary concern. Since the CPU and c-cores do not simultaneously access the cache, the impact on the CPU cycle time is negligible, because the c-cores can mux in through non-critical, pre-existing paths that are used to handle cache fills.

Multi-cycle instructions Conservation cores handle other multi-cycle instructions (e.g., integer division and floating point operations) in the same way as memory operations. Each multi-cycle instruction resides in a basic block with a self-loop and generates a "valid" signal when it completes.

Example Figure 3 shows the translation from C code (a) to hardware schematic and state machine (c). The hardware corresponds very closely to the CFG of the sample code (b). It has muxes for variables *i* and *sum* corresponding to the *phi* operators in the CFG. Also, the state machine of the c-core is almost identical to the CFG, but with additional self-loops for multi-cycle operations. The datapath has a load unit to access the memory hierarchy to read the array *a*.

4.2 The CPU/c-core interface

Aside from the cache, the only connection between the CPU and the c-cores is a set of scan chains that allow the CPU to manipulate all of the c-core's internal state. The CPU side of the interface is shared among all c-cores on the CPU's tile. The CPU can communicate via scan chains with only one c-core at a time, with switching controlled by the CPU. The CPU uses the scan chains to install patches that will be used across many invocations, and to pass initial arguments for individual invocations. The scan chains also allow the CPU to read and modify internal c-core state to implement exceptions.

Conservation core scan chains are divided into two groups: There are a small number of short, fixed-function scan chains for control, argument passing, and patch installation, and up to 32 scan chains for accessing datapath state.

The scan chains for arguments are short (just 64 bits) to make invocation fast, but the patch installation scan chains can be much longer (up to 13,000 bits in our biggest c-core). However, patch installation is infrequent, so the cost of accessing the scan chain is minor. A special "master control" scan chain contains a single bit and allows the CPU to start and stop the c-core's execution as needed.

Datapath scan chains allow the CPU to manipulate arbitrary execution state during an exception. Datapath scan chains range in length from 32 to 448 bits in our largest c-core.

To access the interface, the CPU provides three new instructions: Move-From-ScanChain (MFSC), Move-To-ScanChain (MTSC), and ScanChain-Rotate-Left (SCRL). MFSC moves the 32 bits at the head of a scan chain into a general purpose processor register, and MTSC does the reverse. SCRL rotates a scan chain left by n bits. SCRL executes asynchronously but MFSC and MTSC have blocking semantics: They will wait for previously issued SCRLs on a scan chain to finish before returning a value from that scan chain.

4.3 Initiating c-core execution

When compiling an application or library containing functions that are compatible with a c-core, the compiler will insert stubs that enable the code to choose between using the c-core-enabled version or the CPU-only version at runtime.

Later, when the application is run and calls the function, the stub checks for an available c-core. If it finds one, it uses the scan chain interface to pass arguments to the c-core, starts it running, and then waits for execution to complete. Once execution completes, the c-core raises an exception and control transfers back to the stub, which extracts the return value and passes it back to the caller.

As the c-cores are drop-in replacements for existing code, programs need not block if the correct c-core is not available (i.e., if it is in use by another program or currently configured with the incorrect patch). The original CPU (software) implementation is still available, and the program can use it instead.

4.4 Patching support

Our analysis of successive versions of our applications revealed a number of common change patterns. The most common changes included modifications to hard-coded constants, the removal or addition of code segments, and the replacement of one operation by another (e.g., an add becomes a subtract). Fortunately, many of these changes can be addressed effectively with very modest amounts of reconfigurability. Conservation cores provide three facilities that can be used to adjusted their behavior after they have been fabricated.

Configurable constants We generalize hard-coded immediate values into configurable registers to support changes to the values of compile-time constants and the insertion, deletion, or rearrangement of structure fields.

Generalized single-cycle datapath operators To support the replacement of one operator with another, we generalize any addition or subtraction to an adder-subtractor, any comparison operation to a generalized comparator, and any bitwise operation to a bitwise ALU. A small configuration register is then added for each such operator, determining which operation is currently active.

Control flow changes In order to handle changes in the CFG's structure and changes to basic blocks that go beyond what the above mechanisms can handle, the c-cores provide a flexible exception mechanism. The control path contains a bit for each state transition that determines whether the c-core should treat it as an exception.

When the state machine makes an exceptional transition, the c-core stops executing and transfers control to the general-purpose core. The exception handler extracts current variable values from the c-core via the scan-chain-based interface, performs a portion of the patched execution, transfers new values back into the c-core, and resumes execution. The exception handler can restart c-core execution at any point in the CFG, so exceptions can arbitrarily alter control flow and/or replace arbitrary portions of the CFG.

The next section describes the patch generation algorithm that determines the reconfiguration state necessary to allow a c-core to continue to run code even after it has been changed from the version used to generate that c-core.

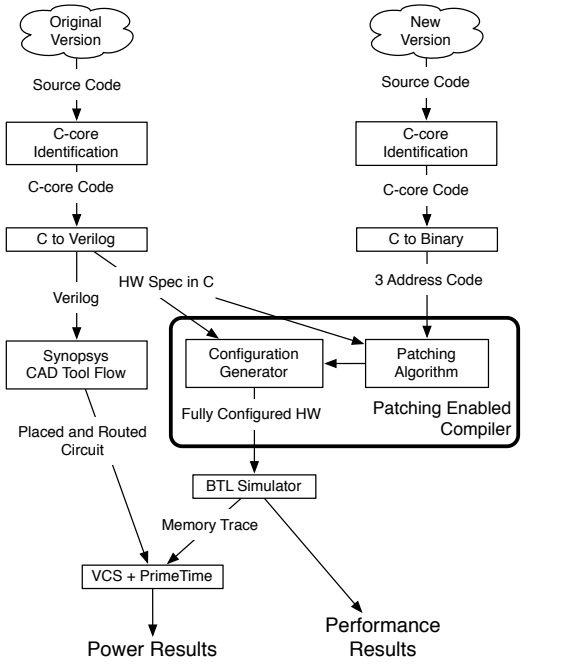

Figure 4. The c-core C-to-hardware toolchain The various stages of our toolchain involved in hardware generation, patching, simulation, and power measurement are shown. The bold box contains the patch generation infrastructure based on our patching enabled compiler.

5. Patching conservation cores

This section describes the patching algorithm we have developed. We expect that at least some patch generation will need to occur at the assembly language level, so our patching algorithm works directly on the program's dataflow and control graphs, a representation that can be generated from either source code or a compiled binary. The bold box in Figure 4 shows how the patching system fits into the toolchain.

When a c-core-equipped processor ships, it can run the latest versions of the targeted applications without modification. We refer to this version as the *original*. When a new version of an application becomes available, we must determine how to map the new version of the software onto the existing c-core hardware. We refer to the new version of the software as the *target*. The goal of the patching process is to generate a *patch* for the original hardware that will let it run the target software version.

Our patching algorithm proceeds in four stages: basic block mapping, control flow mapping, register remapping, and patch generation.

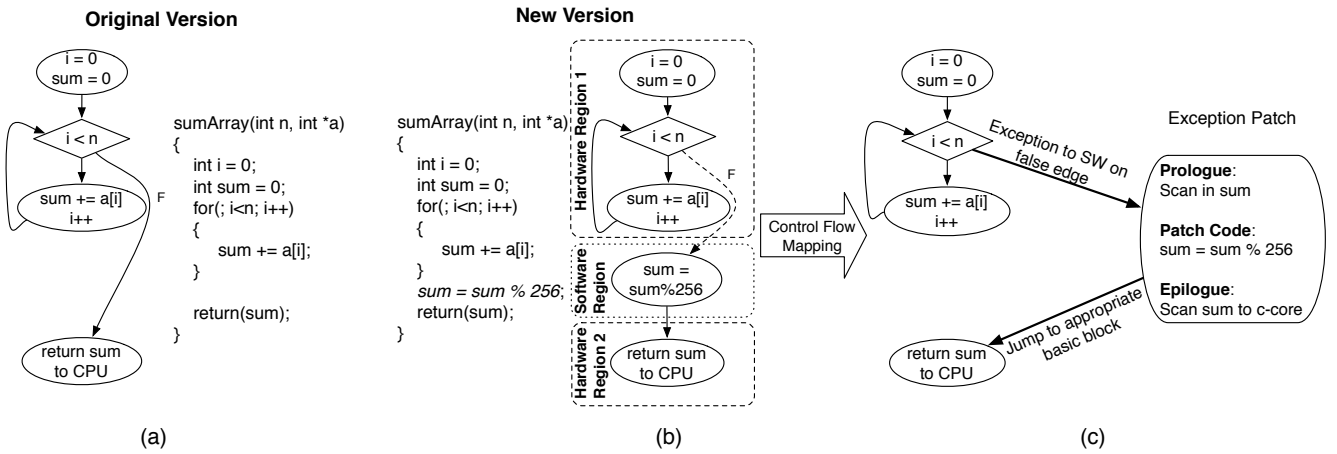

Figure 5. Handling changes in control flow across versions The original source for *sumArray()* is shown alongside its CFG (a). The mapping between the new and the original version of sumArray's CFG covers most of the target version in two hardware regions (b). Transfers of control between the hardware and software regions require an exception (c).

5.1 Basic block mapping

The first stage of the algorithm identifies which hardware basic blocks in the original hardware can run each basic block in the target application. Since the original hardware includes generalized arithmetic operators and configurable constant registers, there is significant flexibility in what it means for two basic blocks to "match". In our system, two basic blocks match if the data flow graphs of the two basic blocks are isomorphic up to operators at the nodes and constant values.

5.2 Control flow mapping

The next step is building a map between the control flow graphs of the original and target versions. We identify regions of the target control flow graph that map perfectly onto disjoint portions of the original hardware. We call these portions of the function *hardware regions*, and they will execute entirely in hardware under the patch. Ideally, all basic blocks in the target will map to basic blocks in the original, and there will be a single hardware region. In practice this will sometimes not be possible: The target version may have basic blocks inserted or deleted relative to the original, or one of the basic blocks may have changed enough that no matching basic block exists in the original. We use the exception mechanism to execute the remaining, unmapped *software regions* on the general purpose processor.

To divide the graph, the algorithm starts by matching the entry node of the target graph with the entry of the original graph. The algorithm proceeds with a breadth-first traversal of the target graph, greedily adding as many blocks to the hardware region as possible. When that hardware region can grow no larger, the region is complete.

A region stops growing for one of two reasons: It may reach the end of the function or run up against another hardware region. Alternatively, there may be no matching basic blocks available to add to the region because of a code modification. In that case, we mark the non-matching basic blocks as part of the software region and select the lowest depth matching basic block available to seed the creation of a new hardware region. This stage of the algorithm terminates when the entire function has been partitioned into hardware regions and software regions.

Figure 5 illustrates this portion of the algorithm. Figure 5(a) shows the original software version of a function called *sumArray()* and its CFG. Figure 5(b) shows the target version of sumArray() which has an extra operation. Most of the new sumArray() is mapped onto the original c-core in two hardware regions, but the new operation is mapped to a separate software region because the hardware for it does not exist in the original c-core. Any transition to this region will be marked as an exception.

5.3 Register mapping

The next phase of the algorithm generates a consistent local mapping between registers in the original and target basic block for each matched basic block pair. In this mapping the output of the first instruction in the original basic block corresponds to the output of the first instruction in the target basic block, and so on.

The next step is to combine these per-block maps to create a consistent register mapping for each hardware region. To construct the mapping, we analyze the basic block mapping for each of the basic blocks in the region. These data yield a weighted bipartite graph, in which one set of nodes corresponds to the register names from the original code and the second set corresponds to register names from the target code. An edge exists between an original register, $r1$, and a target register, $r2$, if the mapping for some pair of basic blocks maps $r2$ onto $r1$. The weight of the edge is the number of pairs of basic blocks whose mapping makes that conversion.

Next, we perform a maximum cardinality, maximum weight matching on the graph. The resulting matching is the register map for that hardware region. Finally, we examine each pair of corresponding basic blocks to see if their register names are consistent with the newly created global map. If they are not, we remove the target basic block from the hardware region and place it in its own software region.

5.4 Patch generation

At this point we have all the information required to generate a patch that will let the target code run on the original hardware. The patch itself consists of three parts:

- the configuration bits for each of the configurable datapath elements along with values for each of the configurable constant registers
- exception bits for each of the control flow edges that pass from a hardware region into a software region
- code to implement each of the software regions

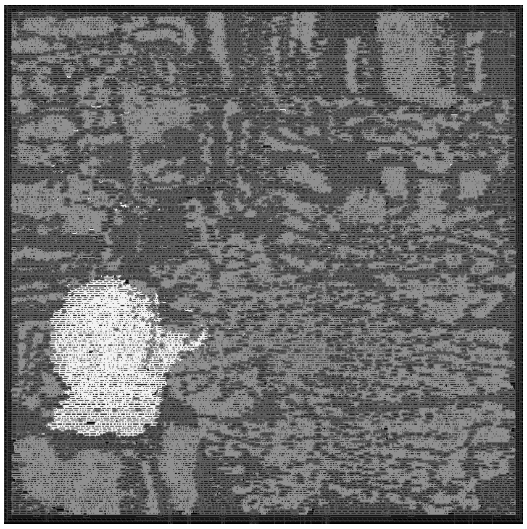

Figure 6. MCF 2006 conservation core for primal_bea_mpp() function The c-core synthesizes to $0.077mm^2$, operates at speeds up to 1412 MHz, and provides 53% coverage for the application. The light gray elements are datapath logic (adders, comparators, etc.), dark gray elements are registers, and the white elements constitute the control path.

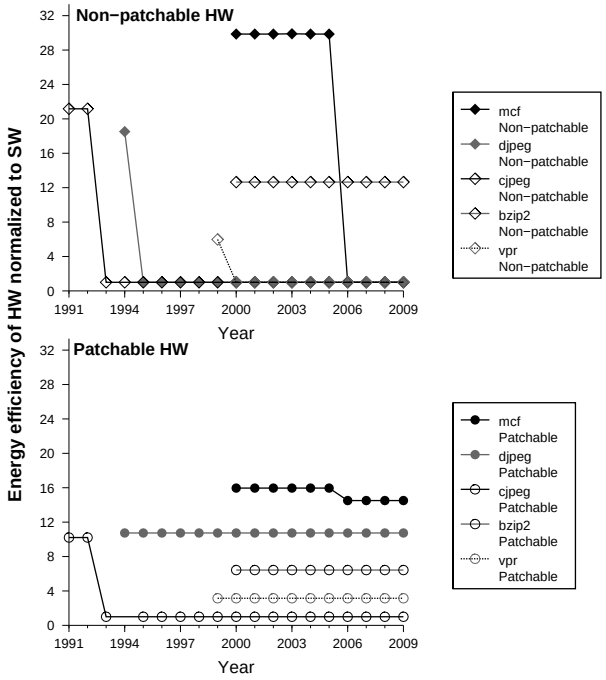

Figure 7. Conservation core effectiveness over time Since c-cores target stable applications, they can deliver efficiency gains over a very long period of time.

The software region code is subdivided into three sections. First, the *prologue* uses the scan chain interface to retrieve values from the c-core's datapath into the processor core. Next, the *patch code* implements the software region. The region may have multiple exit points, each leading back to a different point in the datapath. At the exit, the *epilogue* uses the scan chain interface again to insert the results back into the datapath and return control to the c-core.

5.5 Patched execution example

Figure 5(c) shows how c-cores use the exception mechanism to patch around software regions generated during the control flow mapping stage. When the c-core execution reaches the false edge of the for loop condition, it makes an exceptional transition which freezes the c-core and transfers control to the CPU. The CPU retrieves the application's software exception handler corresponding to the edge that raised the exception, and the handler executes the prologue, patch code, and epilogue before returning control to the c-core.

6. Methodology

Our c-core synthesis toolchain takes C programs as input, splits them into datapath and control segments, and then uses a state-of-the-art EDA tool flow to generate a circuit fully realizable in silicon. The toolchain also generates a cycle-accurate system simulator for the new hardware. We use the simulator for performance measurements and to generate traces that drive Synopsys VCS and PrimeTime simulation of the placed-and-routed netlist. Below, we describe these components in more detail.

6.1 Toolchain

Figure 4 summarizes the c-core toolchain. The toolchain is based on the OpenIMPACT (1.0rc4) [29], CodeSurfer (2.1p1) [8], and LLVM (2.4) [23] compiler infrastructures and accepts a large subset of the C language, including arbitrary pointer references, switch statements, and loops with complex conditions.

In the c-core identification stage, functions or subregions of functions (e.g., key loops) are tagged for conversion into c-cores based on profile information. The toolchain uses outlining to isolate the region and then uses exhaustive inlining to remove function calls. We pass global variables by reference as additional input arguments.

The C-to-Verilog stage generates the control and dataflow graphs for the function in SSA [9] form. This stage then adds basic blocks and control states for each memory operation and multi-cycle instruction. The final step of the C-to-Verilog stage generates synthesizeable Verilog for the c-core. This requires converting ϕ operators into muxes, inserting registers at the definition of each value, and adding self loops to the control flow graph for the multi-cycle operations. Then, it generates the control unit with a state machine that matches the control flow graph. This stage of the toolchain also generates a cycle-accurate module for our architectural simulator.

6.2 Simulation infrastructure

Our cycle-accurate simulation infrastructure is based on *btl*, the Raw simulator [36]. We have modified btl to model a cache-coherent memory among multiple processors, to include a scan chain interface between the CPU and all of the local c-cores, and to simulate the c-cores themselves. The c-cores may operate at different clock frequencies from each other and from the core clock; however the cache clock is synchronized to the c-core when control is transferred to the c-core.

6.3 Synthesis

For synthesis we target a TSMC 45 nm GS process using Synopsys Design Compiler (C-2009.06-SP2) and IC Compiler (C-2009.06-SP2). Our toolchain generates synthesizeable Verilog and automat-

C-core	Ver.	Key	LOC	% Exe.	Area (mm^2) Non-P.	Area (mm^2) Patch.	Freq. (MHz) Non-P.	Freq. (MHz) Patch.
bzip2								
fallbackSort	1.0.0	A *i*	231	71.1	0.128	0.275	1345	1161
fallbackSort	1.0.5	F *i*	231	71.1	0.128	0.275	1345	1161
cjpeg								
extract_MCUs	v1	A *i*	266	49.3	0.108	0.205	1556	916
get_rgb_ycc_rows	v1	A *ii*	39	5.1	0.020	0.044	1808	1039
subsample	v1	A *iii*	40	17.7	0.023	0.039	1651	1568
extract_MCUs	v2	B *i*	277	49.5	0.108	0.205	1556	916
get_rgb_ycc_rows	v2	B *ii*	37	5.1	0.020	0.044	1808	1039
subsample	v2	B *iii*	36	17.8	0.023	0.039	1651	1568
djpeg								
jpeg_idct_islow	v5	A *i*	223	21.5	0.133	0.222	1336	932
ycc_rgb_convert	v5	A *ii*	35	33.0	0.023	0.043	1663	1539
jpeg_idct_islow	v6	B *i*	236	21.7	0.135	0.222	1390	932
ycc_rgb_convert	v6	B *ii*	35	33.7	0.024	0.043	1676	1539
mcf								
primal_bea_mpp	2000	A *i*	64	35.2	0.033	0.077	1628	1412
refresh_potential	2000	A *ii*	44	8.8	0.017	0.033	1899	1647
primal_bea_mpp	2006	B *i*	64	53.3	0.032	0.077	1568	1412
refresh_potential	2006	B *ii*	41	1.3	0.015	0.028	1871	1639
vpr								
try_swap	4.22	A *i*	858	61.1	0.181	0.326	1199	912
try_swap	4.3	B *i*	861	27.0	0.181	0.326	1199	912

Table 3. Conservation core statistics The c-cores we generated vary greatly in size and complexity. In the "Key" column, the letters correspond to application versions and the Roman numerals denote specific functions from the application that a c-core targets. "LOC" is lines of C source code, and "% Exe." is the percentage of execution that each function comprises in the application.

C-core	Ver.	States	Ops	Loads/ Stores	Patching Constructs Add-Sub	Cmp.	Bit.	Const. Reg.	Exc. Bit
bzip2									
fallbackSort	1.0.0	285	647	66 / 38	138	78	33	323	363
fallbackSort	1.0.5	285	647	66 / 38	138	78	33	323	363
cjpeg									
extract_MCUs	v1	116	406	41 / 25	152	11	0	235	127
get_rgb_ycc_rows	v1	23	68	14 / 3	16	2	0	39	25
subsample	v1	32	85	9 / 1	16	8	1	34	40
extract_MCUs	v2	116	406	41 / 25	152	11	0	235	127
get_rgb_ycc_rows	v2	23	68	14 / 3	16	2	0	39	25
subsample	v2	32	85	9 / 1	16	8	1	34	40
djpeg									
jpeg_idct_islow	v5	97	432	39 / 32	180	4	21	238	101
ycc_rgb_convert	v5	40	82	24 / 3	19	4	0	40	44
jpeg_idct_islow	v6	97	432	39 / 32	180	4	21	238	101
ycc_rgb_convert	v6	40	82	24 / 3	19	4	0	40	44
mcf									
primal_bea_mpp	2000	101	144	36 / 16	22	21	0	94	122
refresh_potential	2000	44	70	17 / 5	6	10	0	35	54
primal_bea_mpp	2006	101	144	36 / 16	22	21	0	94	122
refresh_potential	2006	39	60	16 / 4	3	8	0	29	47
vpr									
try_swap	4.22	652	1095	123 / 86	108	149	0	367	801
try_swap	4.3	652	1095	123 / 86	108	149	0	367	801

Table 4. Conservation core details "States" is the number of states in the control path, "Ops" is the number of assembly-level instructions, and "Patching Constructs" gives a breakdown of the different types of patching facilities used in each conservation core.

ically processes the design in the Synopsys CAD tool flow, starting with netlist generation and continuing through placement, clock tree synthesis, and routing, before performing post-route optimizations. We specifically optimize for speed and power.

Figure 6 shows an automatically-generated c-core from the MCF 2006 application. Over 50% of the area is devoted to performing arithmetic operations in the datapath, 7% is dedicated to the control logic, and 40% is registers. This circuit meets timing at clock frequencies up to 1412 MHz.

6.4 Power measurements

In order to measure c-core power usage, our simulator periodically samples execution by storing traces of all inputs and outputs to the c-core. Each sample starts with a "snapshot" recording the entire register state of the c-core and continues for 10,000 cycles. The current sampling policy is to sample 10,000 out of every 50,000 cycles, and we discard sampling periods corresponding to the initialization phase of the application.

We feed each trace sample into the Synopsys VCS (C-2009.06) logic simulator. Along with the Verilog code our toolchain also automatically generates a Verilog testbench module for each c-core, which initiates the simulation of each sample by scanning in the register values from each trace snapshot. The VCS simulation generates a VCD activity file, which we pipe as input into Synopsys PrimeTime (C-2009.06-SP2). PrimeTime computes both the static and dynamic power for each sampling period. We model fine-grained clock gating for inactive c-corestates via post-processing.

To model power for other system components, we derive processor and clock power values from specifications for a MIPS 24KE processor in TSMC 90 nm and 65 nm processes [26], and component ratios for Raw reported in [21]. We have scaled these values for a 45 nm process and assume a MIPS core frequency of 1.5 GHz with 0.077 mW/MHz for average CPU operation. Finally, we use CACTI 5.3 [37] for I- and D-cache power.

7. Results

This section describes the performance and efficiency of our c-core-enabled architecture and the impact of c-cores on application performance and energy consumption. Then we analyze the overheads due to patching and the potential benefits of applying c-cores to a wide range of applications.

7.1 Energy savings

Figure 8 shows the relative energy efficiency, EDP improvement, and speedup of c-cores vs. a MIPS processor executing the same code. For fairness, and to quantify the benefits of converting instructions into c-cores, we exclude cache power for both cases. The figure compares both patchable and non-patchable c-cores to a general-purpose MIPS core for six versions of bzip2 (1.0.0−1.0.5), and two versions each of cjpeg (v1−v2), djpeg (v5−v6), mcf (2000−2006), and vpr (4.22−4.30). Tables 3 and 4 summarize the c-cores.

The data show that patchable c-cores are up to 15.96× as energy-efficient as a MIPS core at executing the code they were built to execute. The non-patchable c-cores are even more energy efficient, but their inability to adapt to software changes limits their useful lifetime.

Figure 7 quantifies the ability of patchability to extend the useful lifetime of c-cores. The horizontal axis measures time in years, and the vertical axis is energy efficiency normalized to software. The lines represent what the c-cores built for the earliest software version can deliver, both with and without patching support. For instance, vpr 4.22 was released in 1999, but when a new version appears in 2000, the non-patchable hardware must default to software, reducing the energy efficiency factor from 6.2× down to 1×. In contrast, patchable hardware built for 4.22 has a lower initial efficiency factor of 3.4×, but it remained there until March 2009. For djpeg and bzip2, the results are even more impressive: Those c-cores deliver 10× and 6.4× energy efficiency improvements for covered execution over 15 and 9 year periods, respectively.

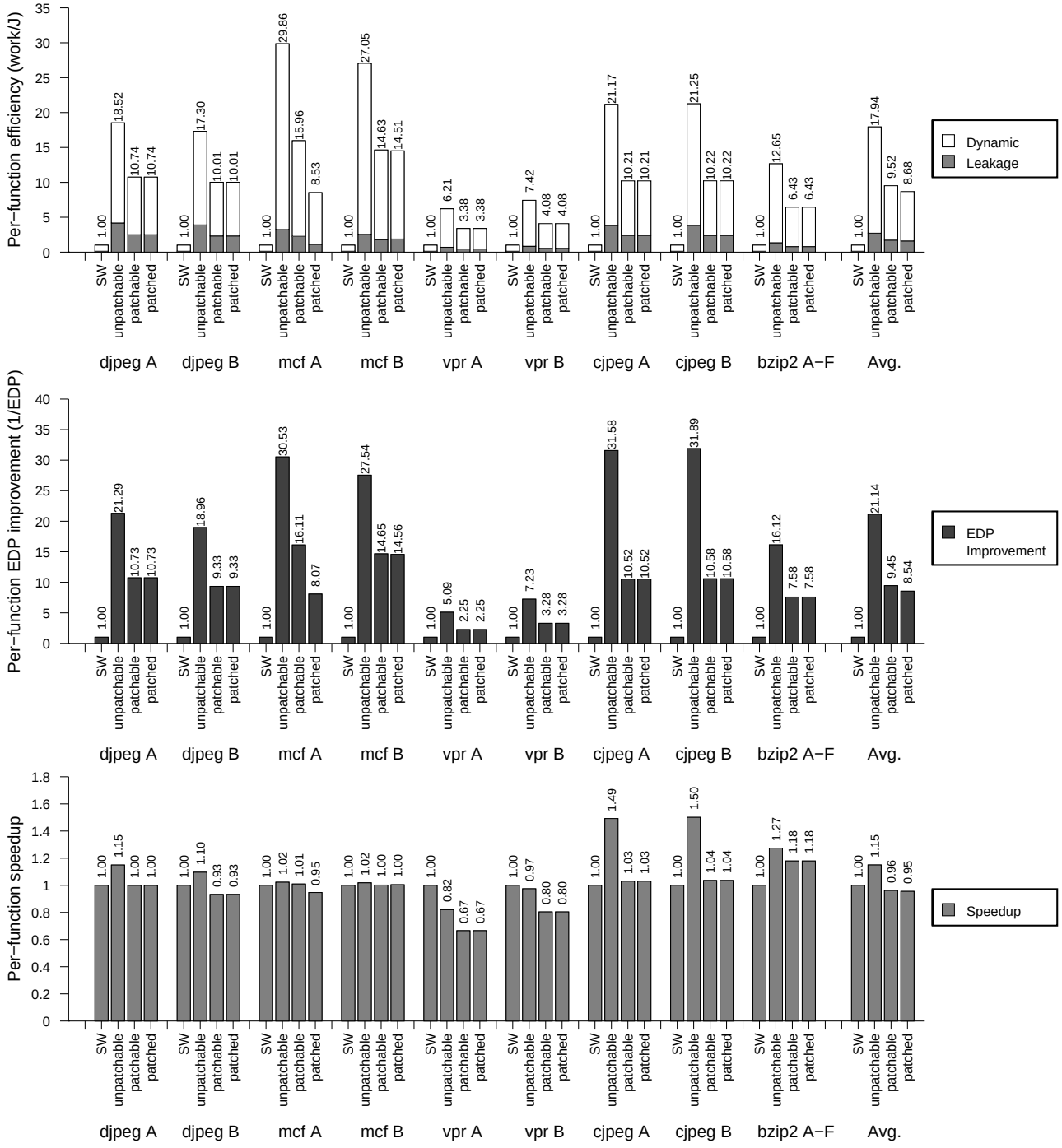

Figure 8. Conservation core energy efficiency Our patchable c-cores provide up to 15.96× improvement in energy efficiency compared to a general-purpose MIPS core for the portions of the programs that they implement. The gains are even larger for non-patchable c-cores, but their lack of flexibility limits their useful lifetime (see Figure 7). Each subgroup of bars represents a specific version of an application (see Table 3). Results are normalized to running completely in software on an in-order, power-efficient MIPS core ("SW"). "unpatchable" denotes a c-core built for that version of the application but without patching support, while "patchable" includes patching facilities. Finally, "patched" bars represent alternate versions of an application running on a patched c-core. For all six versions of bzip2 (A-F), the c-core's performance is identical.

Structure	Area (μm^2)	Replaced by	Area (μm^2)
adder subtractor	270 270	AddSub	365
comparator (GE)	133	Compare6	216
bitwise AND, OR bitwise XOR	34 56	Bitwise	191
constant value	~ 0	32-bit register	160

Table 5. Area costs of patchability The AddSub unit can perform addition or subtraction. Similarly, Compare6 replaces any single comparator (e.g., $\geq$) with any of ($=, \neq, \geq, >, \leq, <$). Constant values in non-patchable hardware contribute little or even "negative" area because they can enable many optimizations.

7.2 Patching overhead

Although patching provides significant benefits, it incurs additional overhead. Below we examine the impacts of patching on area and energy consumption in more detail.

Area overhead Patching area overhead comes in four forms. The first is the increase in area caused by replacing simple, fixed-function datapath elements (e.g., adders, comparators) with configurable ones. The second form comes from the conversion of hardwired constant values to configurable registers. Third, there is extra area in the scan chains that allow us to insert, remove, and modify arbitrary values in the c-core. Finally, patchable c-cores require additional area in the control path to store edge exception information.

Table 5 compares the area requirements for patchable and non-patchable structures. The costs on a per-element basis vary from 160 μm^2 per configurable constant register to 365μm^2 per add/subtract unit. The standard cell libraries we use include scan chains in registers automatically, but they are not easily accessible in the tool flow. Instead, we implement the scan chains explicitly, which results in additional overhead that could be removed with proper support from the tool flow.

Figure 9(a) shows the breakdown of area for the patchable c-cores for the earliest version of each of our 5 target applications. Patching support increases the area requirements of c-cores by 89% on average.

Power overhead Patching support also incurs additional power overhead. Figure 9(b) shows the impact of adding each of the three components described above. Overall, patching support approximately doubles the power consumption of the c-cores, with the majority of the overhead coming from the configurable registers.

Performance overhead In our toolchain, adding patchability does not change the structure of the datapath, but it does potentially increase the critical path length and, therefore, the achievable clock speed. On average, patchable systems achieve 90% of the application performance of non-patchable systems.

Reducing overheads We can reduce the overhead of patching by performing a more detailed analysis of changes in stable workloads. If it were possible to identify, for instance, which constants or operators are more likely to change, we could provide patchability only where it is likely to be useful. This would reduce area costs, reduce dynamic and static power consumption, and improve performance.

7.3 Examining Amdahl's Law

Figures 7 and 8 demonstrate that c-cores can provide large efficiency gains for individual functions throughout a chip's useful lifetime. In this section, we examine the impact of Amdahl's law by evaluating full-application, full-chip c-core energy efficiency. Figure 10 shows the energy (top), energy-delay product (middle), and delay (bottom) for each of the applications. In each group, the first bar measures the performance of the program without using any c-cores. The second measures performance with the patchable c-cores described in the previous section. The data show that, at the application level, c-cores save between 10% and 47% of energy compared to running on a MIPS core, and they reduce EDP by between 1% and 55%. Runtime varies, decreasing by up to 12% and increasing by up to 22%, but only increasing 1% on average.

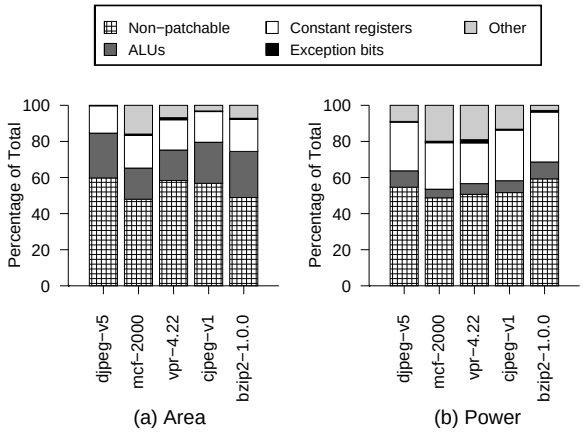

Figure 9. Area and power breakdown for patchable c-cores Adding patchability approximately doubles a c-core's area (a) and power (b) requirements.

While the benefits for these c-core-enabled systems are sizable, the efficiency improvements from c-cores are moderated by the fact that the remaining parts of the system are largely untuned for use in this context. There are two key ways to realize larger full-application benefits from c-cores that appear particularly promising for future work. We explore the potential of both of these options below.

Increasing coverage The first, and most effective, is to increase the fraction of execution spent on a c-core, or coverage. The amount of coverage that c-cores can deliver is a function of the applications the system targets, the c-core toolchain, and the overheads that c-cores introduce.

For c-cores to achieve high coverage in a reasonable amount of area, a relatively small fraction of the application's static instructions must account for a large fraction of execution. Figure 11 shows the fraction of dynamically executed x86 instructions (y-axis) covered by the number of static x86 instructions (x-axis) for a broad-ranging set of applications, including SPECCPU2006 integer benchmarks astar and hmmer, desktop applications evince, emacs, grep, gcc, perl, and scp, and the five applications for which we constructed c-cores. For many applications, the curve is quite steep. This means that converting a relatively small number of static instructions into hardware will cover a very large fraction of execution.

The third bar in each subgroup in Figure 10 shows our projections assuming that we can achieve a 90% degree of coverage, which Figure 11 indicates should be possible with a hardware budget covering less than 5000 static instructions per application for the applications in question. Energy savings would then range from 40% to 62%, and EDP savings from 18% to 63%.

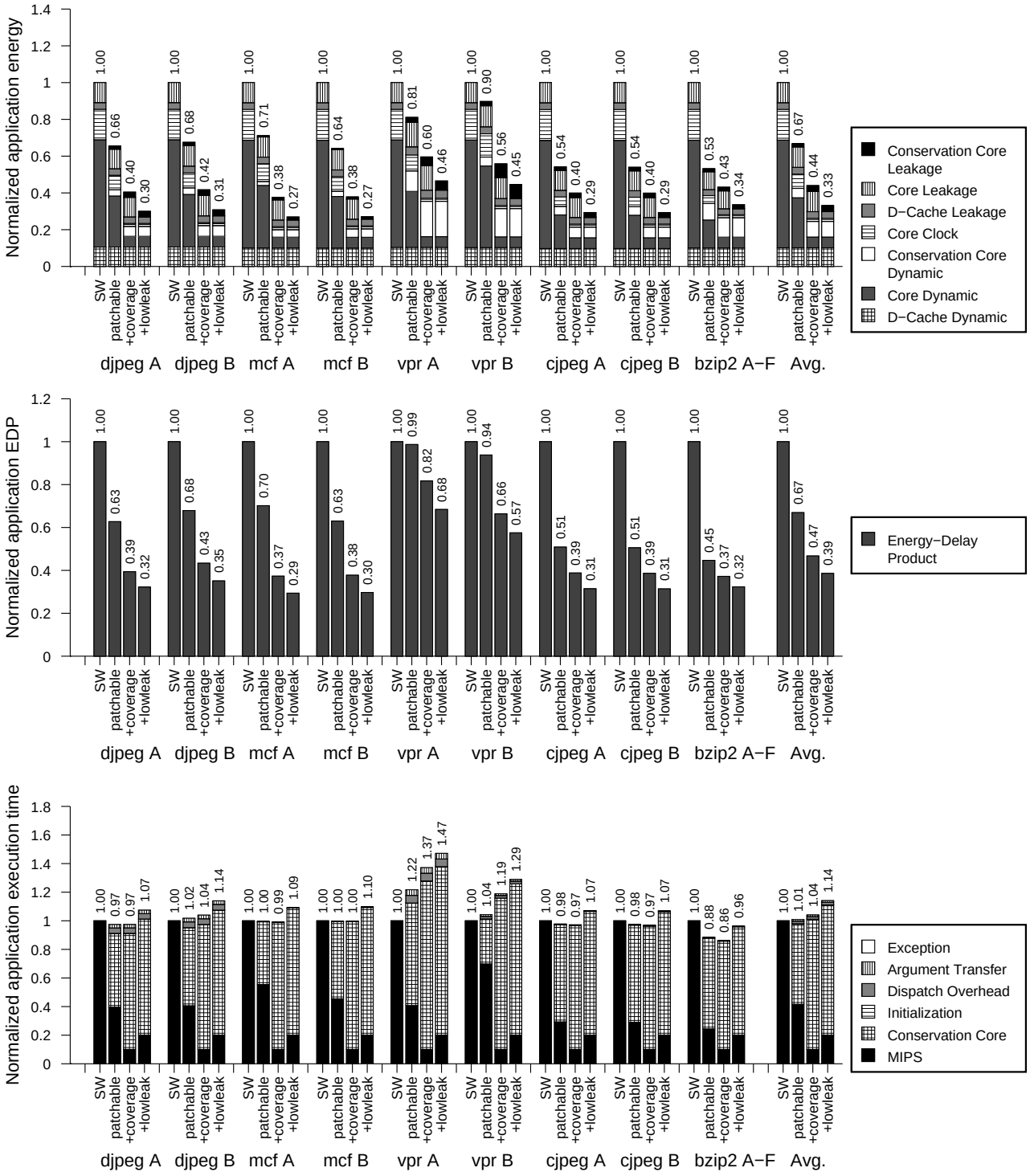

Figure 10. Full application system energy, EDP, and execution time for c-cores, and projected improvements These graphs show full application system energy, EDP, and execution time for c-cores (lower is better). "SW" and "patchable" are as described in Figure 8. "+coverage" displays achievable improvements to energy reduction if 90% of the application can run in a c-core. If a slower, lower-leakage process is used for the MIPS core in addition to improved coverage ("+lowleak"), even further improvements are possible. As in Figure 8, each subgroup of bars represents a specific version of an application, and for all six versions of bzip2 the c-core's performance is identical.

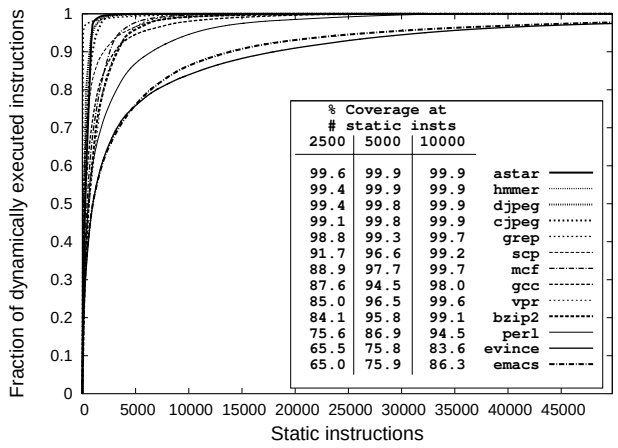

Figure 11. Dynamic coverage for given static instruction counts For many x86 programs profiled, a small number of static instructions cover much of dynamic execution.

Several limitations of our toolchain currently prevent us from achieving this level of coverage. One is our lack of support for floating point functions such as *exp* and *sqrt*. For instance, adding better floating point support would allow us to increase the coverage for vpr 4.30 from 27% to 54% improving energy savings.

Reducing fixed costs The second approach is to reduce the energy consumption of the other components in the system. As c-core coverage increases, the relative importance of the performance of the general-purpose CPU goes down. This changes the performance/efficiency balance point for the CPU, and calls for changes to the MIPS core. For instance, the contribution of the processor to leakage could be greatly reduced by switching to high-V_t transistors. With higher c-core coverage, this would be quite sensible, as the performance penalties to the processor would only affect small portions of execution (i.e., 10%). This same approach can be applied to the instruction cache and other peripheral components. Shared components still heavily used by c-cores, however, such as the data cache, could not be as aggressively modified without compromising performance.

The final bar in each group in Figure 10 shows the impact of making these changes to the other system components. Rebalancing reduces the fixed cost overheads of instruction cache and processor leakage at the cost of approximately halving the performance of the software component of execution. Reducing these fixed costs provides an additional 11% savings in energy and increases the total energy savings to 67% and total EDP savings to 61% on average.

8. Related Work

A vast number of specialized processors have been proposed, many of which provide acceleration for very narrowly defined workloads. Specialized designs exist for applications and domains such as cryptography [39], signal processing [12, 16], vector processing [3, 10], physical simulation [2], and computer graphics [4, 28, 30]. A range of commercial heterogeneous processors and research prototypes are available or have been proposed. These include Sony's Cell [20], IRAM [31], and Intel's EXOCHI [13]. These machines augment a general purpose core with vector co-processors to accelerate multimedia applications. Other vendors, such as Phillips [32] and Equator [25], provide non-configurable but specialized designs for media applications. The work in [15] and [41] provide customized, yet flexible circuits by building programmable, specialized loop accelerators and merging multiple circuits into one, respectively.

In contrast to many of the above examples, our approach is general, rather than domain-specific. We can target any C codebase once it reaches a minimal level of code stability, and our patching mechanisms allow continued utility from c-core hardware even as codebases change. Additionally, unlike most of the above designs, we prioritize energy reduction over performance improvement.

Other approaches to hardware specialization (e.g., Tensilica's Stenos [38], OptimoDE [7], and PICO [1]) provide easy-to-modify designs that customers customize either before or after manufacturing. Our patch-based post-manufacturing capabilities are more flexible than hardware customization.

Strozek and Brooks [34] improve the energy efficiency for a set of applications in the embedded space by automatically selecting specialized cores from a well-defined design space. The cores are Pareto-optimal for the target workloads. Our automated approach admits a much larger range of core designs, but sacrifices formal guarantees of optimality.

Previous work has proposed combining cores that exhibit microarchitectural heterogeneity to improve performance or reduce power consumption on general purpose workloads, and many commercial products target a small class of applications with a highly tailored set of co-processors. Designs such as Chimaera [40], GARP [18], PRISC [33], and the work in [6] augment a general-purpose processor with reconfigurable logic. While the c-core patching mechanism does provide a degree of reconfigurability and associated overheads, the application-specific nature of a c-core still yields an energy efficiency much closer to an ASIC than to a reconfigurable fabric or co-processor.

Recent work on single-ISA heterogeneous multi-core processors [5, 14, 17, 22, 24] investigates the power and performance trade-offs for CMPs with non-uniform cores. They use phase-transition-driven partitioning to trade 10% of performance for a nearly 50% reduction in power consumption by moving execution between aggressive out-of-order cores and simpler, in-order cores. Conservation core architectures can deliver even larger energy savings on top of that, as they can reduce energy consumption by up to 47% over even a simple, *in-order* core.

9. Conclusion

As we run up against the utilization wall, we enter a regime in which reducing energy per operation becomes increasingly important. We have described conservation cores, a new class of circuits that aim to increase the energy efficiency of mature applications. Our toolchain synthesizes c-cores from C code and builds in support that allows them to evolve when new versions of the software appear. Our data for 18 fully placed-and-routed c-cores show that they can reduce energy consumption by 10-47% and energy-delay by up to 55%. More important, we show that c-cores can provide enough flexibility to ensure that they will remain useful for up to 15 years, far beyond the expected lifetime of most processors. As our toolchain matures and coverage improves, we have shown that c-cores have the potential to become even more effective, with projected energy savings of 67%, and EDP savings of 61%. To fully evaluate the potential of Conservation Cores, we are in the process of implementing a prototype c-core-enabled system in STMicro's 45 nm technology.

Acknowledgments

This research was funded by the US National Science Foundation under NSF CAREER Awards 06483880 and 0846152, and under NSF CCF Award 0811794. We would like to thank the anonymous reviewers and Simha Sethumadhavan for their helpful suggestions.

References

[1] S. Aditya, B. R. Rau, and V. Kathail. Automatic architectural synthesis of VLIW and EPIC processors. In *ISSS '99: Proceedings of the 12th international symposium on System synthesis*, page 107. IEEE Computer Society, 1999.

[2] Ageia Technologies. PhysX by Ageia. http://www.ageia.com/pdf/ds_product_overview.pdf.

[3] J. H. Ahn, W. J. Dally, B. Khailany, U. J. Kapasi, and A. Das. Evaluating the Imagine Stream Architecture. In *ISCA '04: Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 14–25. IEEE Computer Society, 2004.

[4] ATI website. http://www.ati.com.

[5] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 506–517, Washington, DC, USA, 2005. IEEE Computer Society.

[6] N. Clark, J. Blome, M. Chu, S. Mahlke, S. Biles, and K. Flautner. An architecture framework for transparent instruction set customization in embedded processors. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 272–283. IEEE Computer Society, 2005.

[7] N. Clark, H. Zhong, K. Fan, S. Mahlke, K. Flautner, , and K. V. Nieuwenhove. OptimoDE: Programmable accelerator engines through retargetable customization. In *HotChips*, 2004.

[8] CodeSurfer by GrammaTech, Inc. http://www.grammatech.com/products/codesurfer/.

[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35. ACM Press, 1989.

[10] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J.-H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, and U. J. Kapasi. Merrimac: Supercomputing with streams. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 35. IEEE Computer Society, 2003.

[11] R. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. In *IEEE Journal of Solid-State Circuits*, October 1974.

[12] C. Ebeling, D. C. Cronquist, and P. Franklin. RaPiD - reconfigurable pipelined datapath. In *FPL '96: Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, pages 126–135. Springer-Verlag, 1996.

[13] P. W. et al. Exochi: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 156–166, New York, NY, USA, 2007. ACM Press.

[14] R. K. et al. Core architecture optimization for heterogeneous chip multiprocessors. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 23–32, New York, NY, USA, 2006. ACM Press.

[15] K. Fan, M. Kudlur, G. Dasika, and S. Mahlke. Bridging the computation gap between programmable processors and hardwired accelerators. In *HPCA: High Performance Computer Architecture.*, pages 313–322, Feb. 2009.

[16] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer. PipeRench: A Coprocessor for Streaming Multimedia Acceleration. In *ISCA '99: Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 28–39. IEEE Computer Society, 1999.

[17] E. Grochowski, R. Ronen, J. Shen, and H. Wang. Best of both latency and throughput. In *ICCD '04: Proceedings of the IEEE International Conference on Computer Design (ICCD'04)*, pages 236–243, Washington, DC, USA, 2004. IEEE Computer Society.

[18] J. R. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In K. L. Pocek and J. Arnold, editors, *FCCM '97: IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12–21. IEEE Computer Society Press, 1997.

[19] M. Horowitz, E. Alon, D. Patil, S. Naffziger, R. Kumar, and K. Bernstein. Scaling, Power, and the Future of CMOS. In *IEDM '05: IEEE International Electron Devices Meeting*, 2005.

[20] J. Kahle. The CELL processor architecture. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, page 3. IEEE Computer Society, 2005.

[21] J. S. Kim, M. B. Taylor, J. Miller, and D. Wentzlaff. Energy characterization of a tiled architecture processor with on-chip networks. In *International Symposium on Low Power Electronics and Design*, San Diego, CA, USA, August 2003.

[22] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *ISCA '04: Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 64. IEEE Computer Society, 2004.

[23] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75. IEEE Computer Society, 2004.

[24] J. Li and J. F. Martínez. Power-performance considerations of parallel computing on chip multiprocessors. *ACM Trans. Archit. Code Optim.*, 2(4):397–422, 2005.

[25] MAP-CA datasheet, June 2001. Equator Technologies.

[26] MIPS Technologies. MIPS Technologies product page. http://www.mips.com/products/processors/32-64-bit-cores/mips32-24ke , 2008-2009.

[27] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. C. Goldstein, and M. Budiu. Tartan: evaluating spatial computation for whole program execution. *SIGOPS Oper. Syst. Rev.*, 40(5):163–174, 2006.

[28] nVidia website. http://www.nvidia.com.

[29] OpenImpact Website. http://gelato.uiuc.edu/.

[30] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, , and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005.

[31] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent RAM. *IEEE Micro*, 17(2):34–44, April 1997.

[32] TM1000 preliminary data book, 1997. http://www.semiconductors.philips.com/acrobat/other/tm1000.pdf.

[33] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, pages 172–180. ACM Press, 1994.

[34] L. Strozek and D. Brooks. Efficient architectures through application clustering and architectural heterogeneity. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 190–200, New York, NY, USA, 2006. ACM Press.

[35] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. J. Eggers. The wavescalar architecture. *ACM Trans. Comput. Syst.*, 25(2):4, 2007.

[36] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *ISCA '04: Proceedings of the 31st annual International Symposium on Computer Architecture*, page 2. IEEE Computer Society, 2004.

[37] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. Cacti 5.1. Technical Report HPL-2008-20, HP Labs, Palo Alto, 2008.

[38] A. Wang, E. Killian, D. Maydan, and C. Rowen. Hardware/software instruction set configurability for system-on-chip processors. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 184–188. ACM Press, 2001.

[39] L. Wu, C. Weaver, and T. Austin. Cryptomaniac: A fast flexible architecture for secure communication. In *ISCA '01: Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 110–119. ACM Press, 2001.

[40] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit. In *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 225–235. ACM Press, 2000.

[41] S. Yehia, S. Girbal, H. Berry, and O. Temam. Reconciling specialization and flexibility through compound circuits. In *HPCA 15: High Performance Computer Architecture*, pages 277–288, Feb. 2009.