Using a Configurable Processor Generator for Computer Architecture Prototyping

Alex Solomatnikov*, Amin Firoozshahian*
Hicamp Systems, Inc.
{solomatnikov, aminf13}@gmail.com

Ofer Shacham, Zain Asgar, Megan Wachs, Wajahat Qadeer, Stephen Richardson, Mark Horowitz
Stanford University
{shacham, zasgar, wachs, wqadeer, steveri, horowitz}@stanford.edu

ABSTRACT
Building hardware prototypes for computer architecture research is challenging. Unfortunately, development of the required software tools (compilers, debuggers, runtime) is even more challenging, which means these systems rarely run real applications. To overcome this issue, when developing our prototype platform, we used the Tensilica processor generator to produce a customized processor and corresponding software tools and libraries. While this base processor was very different from the streamlined custom processor we initially imagined, it allowed us to focus on our main objective—the design of a reconfigurable CMP memory system—and to successfully tape out an 8-core CMP chip with only a small group of designers. One person was able to handle processor configuration and hardware generation, support of a complete software tool chain, as well as developing the custom runtime software to support two different programming models. Having a sophisticated software tool chain not only allowed us to run more applications on our machine, it once again pointed out the need to use optimized code to get an accurate evaluation of architectural features.

Categories and Subject Descriptors
C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors) – *Parallel Processors*; C.4 [**Performance of Systems**]: *Design studies*; C.1.2 [**Computer System Implementation**]: *VLSI Systems*

General Terms
Performance, Design, Experimentation

Keywords
Reconfigurable architecture, configurable/extensible processor generator, memory system architecture, computer architecture prototyping, VLSI design

1. INTRODUCTION AND MOTIVATION
Our expectations of the capability of software development environments continue to grow with time. Compilers, runtime environments, debuggers and performance analysis tools are critical for creating high-performance applications. As Kunz showed in [23], eliminating the small accidental sharing that happens in most parallel applications can have a much larger effect on application performance than most architectural techniques—removing sharing is always better than trying to make sharing more efficient. This fact is a challenge for researchers working on innovative architectures. Detection and debugging of false sharing and other software issues requires a strong set of software tools that are hard and time consuming to develop. Indeed, all computer architecture projects face a dilemma: any architecture that is ambitious and truly novel by definition requires enormous software development effort to make it useable; on the other hand, architectures that can mostly reuse existing software tools are often incremental in nature. As result, many research projects that try to explore novel architectural ideas end up spending significant effort on software: RAW [46], Imagine [3][21], TRIPS [13] and TCC [16] are all recent examples. However, even very significant software effort does not guarantee that the performance of the generated software is good enough to show the advantages of a new architecture, and as a result researchers often resort to hand-optimization of benchmarks [13].

We recently completed an implementation of the Smart Memories CMP, which faced similar challenges [38]. The goal of the Smart Memories project was to design a reconfigurable architecture that could support a wide variety of memory models. To test its flexibility we implemented conventional cache-coherent shared memory [1], streaming [3][21] and transactions [16][17]. Initially we planned to have reconfigurability in both the processor and the memory system. After working on this path for a while, we realized that the development of a new machine and instruction set architecture along with the associated software stack was going to be much too large a task for the small group of students involved with the project. Instead we decided to concentrate on memory system architecture and took the unusual path of trying to leverage the Tensilica system and its associated software tools [14] for the processor portion of our research architecture implementation. This approach let us focus on the core feature of the project—reconfigurable memory system—and to successfully complete the test chip design, with minimal effort (one person) devoted to the configuration and generation of the processor hardware and software tools. While the use of Tensilica limited our flexibility, it provided a strong software development environment and greatly increased confidence in our architectural

MICRO'09, December 12–16, 2009, New York, NY, USA.

*This work was done while the authors were at Stanford University

performance evaluation.

However, adapting the Tensilica processor to our architecture was challenging. As we describe later in the paper there were many issues to resolve, but the biggest disconnect was the fact that at the time we started, Tensilica did not have a method of building a coherent memory system, and we wanted our memory system to support sophisticated memory operations that required coherence. Fortunately, the extensibility of the Tensilica platform allowed us to implement the range of memory operations that we needed, and the accompanying software system allowed us to create a number of applications to evaluate our new hardware features.

This paper reviews the history of Smart Memories development, the decisions made during this process and the lessons learned, as well as the current status and results. We feel this story makes a compelling case for using extensible processors in future architecture research. To understand the advantages of this approach, the next section reviews how other large computing systems research projects have addressed the software development question. With this background, Section 3 then briefly talks about the history of the Smart Memories project and reviews the key architectural features needed to support our flexible memory system. We faced several challenges when we switched to a Tensilica processor, so Section 4 first provides an overview of the Tensilica environment, and then Section 5 describes how we connected the processor to the flexible memory system, and how we used register windows to quickly restore shadow state. Section 5 also discusses how using an optimizing software pipeline changed some of our performance results. Our concluding remarks are presented in Section 6.

2. RELATED WORK

From MIPS to TRIPS, many architectural research projects developed custom instruction set architectures that required not only the development of new compilers but new compiler technologies. Early RISC projects at Berkeley and Stanford developed the ISA in parallel with compiler and operating system work. These efforts led to many advanced compiler techniques including instruction scheduling and register allocation [8][36]. VLIW architectures further pushed instruction scheduling and relied heavily on optimizing compilers to achieve high performance [11][18][22][28][39]. Similarly, dataflow architectures required sophisticated software systems [4][44]. The TRIPS project is the latest project in this vein, which designed the EDGE architecture along with the superblock scheduling compiler that it required [40].

Some of the most innovative machines from an architectural standpoint require more than just a custom compiler, since they are based on a different programming model. Some examples include systolic computing by CMU's iWarp [7], message driven computing by MIT's J-Machine [35], stream processing by MIT Raw [46] and Stanford's Imagine [21]. Other projects have adopted existing instruction set architectures (ISA) while exploring multi-processor architectures. For example, Stanford DASH and FLASH used commercial off-the-shelf MIPS processors and designed multi-processor memory systems connected to the processor via a standard processor bus [26][24]. Other projects that used existing ISA or processor designs are Raksha [20] and Wisconsin Wind Tunnel [34].

An alternative approach is to use an existing processor architecture, but modify or add some instructions to let it access new architectural features. An early example is the Sparcle processor, which was developed and used for the MIT Alewife project [2]. Recently there have been a number of companies that have created extensible microprocessor IP, including Tensilica [14] [49][19], MIPS, and ARM. Other companies have created systems that generate processors and software systems from a high level ISA description. These solutions have the potential to provide the advantages of a high-quality software environment while preserving the ability to modify the processor architecture. Given the small design team in the Smart Memories project, we decided that this was the only feasible way to accomplish our goals. While this approach is not perfect, as the rest of the paper shows, it has significant advantages. The following sections describe our experience in the Smart Memory project using this approach.

3. SMART MEMORIES, VERSION I

Recently, with the shift towards multi-core chips, computer architecture research has focused on parallel architectures, as well as the programming models and software systems necessary to facilitate the wide adoption of such architectures. The goal of the Smart Memories project was to design a universal programming platform that can support three popular programming models that were explored by other researchers. One of these is the conventional cache-coherent shared memory model [1]. Another is the stream programming model implemented by the MIT RAW and Stanford Imagine [45][46][3][21]. These projects developed streaming languages [48], associated software tool chains and even application benchmarks written using the stream programming model. Transactional memory was another memory model that we wanted to support [17][16]. The transactional memory programming model was proposed to simplify parallelization of software using a hardware transactional mechanism, which builds upon work on thread-level speculation (TLS) [15]. This work involves designing an application programming interface (API) for transactional memory hardware and runtime system [32][5] and developing application benchmarks that can be used to evaluate transactional memory architectures [33]. Initially, the architecture was designed to support TLS. Later in the implementation process we decided instead to complete the implementation of a more general transactional memory model, transactional coherence and consistency (TCC) [16].

To provide functional flexibility, Smart Memories consists of a hierarchical reconfigurable architecture. The basic building blocks include a reconfigurable memory system, plus interconnect and processing elements [29]. The chip is divided into tiles, and four adjacent tiles are grouped into quads (Figure 1a). Quads communicate with each other and with off-chip interfaces via a dynamically routed global on-chip network. Tiles inside the quad share the network port and protocol controller. Each tile (Figure 1b) contains 16 memory blocks, which are connected to the processor through a crossbar. Each memory block (Figure 1c) is also reconfigurable [30].

In addition to an SRAM data array, memory blocks also have a dual-ported metadata array that stores control state associated with the data and some programmable logic that enables a set of customizable atomic read-modify-write operations for this control state. Each memory block also contains a comparator, so it can be used as tag memory, and FIFO pointers. Using these features and a flexible crossbar interconnect, a set of memory blocks in the tile

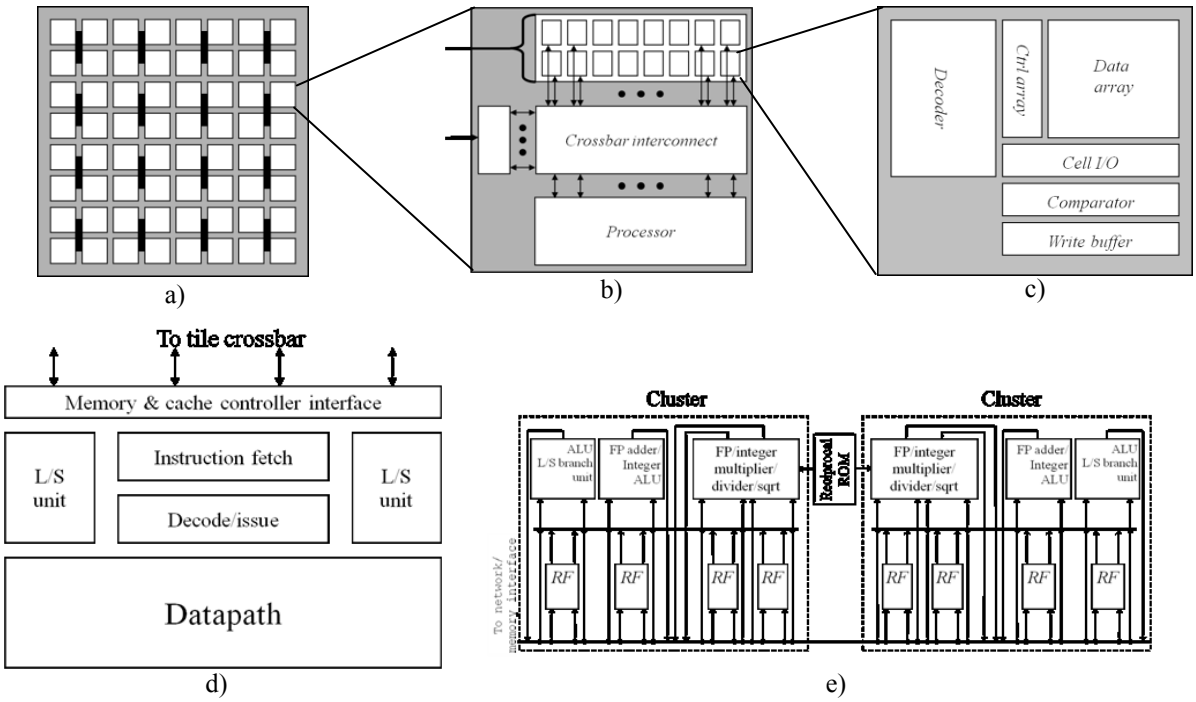

Figure 1. Smart Memories Architecture: a) chip floorplan; b) tile block diagram; c) reconfigurable memory block diagram; d) reconfigurable processor; e) shared processor data-path

can be configured to function as set-associative caches, directly addressable local memories (scratchpads) or as hardware FIFOs. The quad protocol controller performs all actions required for data movement functionality such as cache line refills, streaming DMAs or transaction ordering. It is also reconfigurable to support different modes of operation of the memory system such as cache coherence or thread-level speculation.

In its original version, the processor (Figure 1d) was also designed to be configurable, with three modes of operation. In multi-threaded mode the processor behaves as 2 processors, each running independent threads of instructions, and each thread executing up to 2 instructions per cycle. Multi-threaded mode would be a good match for applications with abundant thread-level parallelism (TLP). In VLIW mode the processor would execute a single instruction stream with up to 4 instructions per cycle, which would fit applications with a significant amount of instruction-level parallelism (ILP). Finally, to achieve highest possible performance (up to 10 operations per cycle) in streaming mode, each unit of the data-path and each port of each register file (Figure 1e) would be controlled explicitly by a very wide 256-bit instruction, similar to the Stanford Imagine processor [21]. The data-path (Figure 1e) was designed to support all three modes of operations by changing the routing of operation operands and results.

At this point we realized that we had been focusing on the smaller part of the problem. While this architecture had a number of interesting features, and the hardware was modular and seemed like it would not be too difficult to implement on a test-chip, it became clear that the effort required to develop the corresponding software infrastructure was too high. The three processor modes required development of three different software development environments. Each had a different instruction set (ISA) and computational model, so each mode would require its own compiler and support tools such as assemblers, linkers, debuggers, runtimes and IO libraries. In addition, the design of a processor with different custom instruction sets would require three sets of ISA verification tests as well as development or tuning of software benchmarks. This was impossible for a small team at a university.

With this heightened awareness of the costs of the software environment and testing, we needed to find a feasible solution. A conventional CPU would not support novel memory operations, thus the need for some degree of customization. At the minimum, it appeared that we would need at least a few special processor features, including (1) different flavors of memory operations and (2) the ability to recover from speculative operations. Because Tensilica presented a solution that supports custom features yet does not require ground-up hardware and software design [14], we decided to try and integrate it into our system.

Using Tensilica reduced the amount of required software work substantially and allowed us to focus on the design of a reconfigurable memory system [12]. Obviously, it also constrained the Smart Memories architecture and potentially limited its performance. This was a small price to pay to complete the design with a reasonable amount of effort and to have a high-quality optimizing compiler and other software tools.

4. TENSILICA OVERVIEW

Tensilica's Xtensa Processor Generator automatically generates a synthesizable hardware description for the user customized processor configuration [14][25]. The base Xtensa architecture is a 32-bit RISC instruction set architecture (ISA) with 24-bit instructions and a windowed general-purpose register file. Register windows have 16 register each, and the total number of

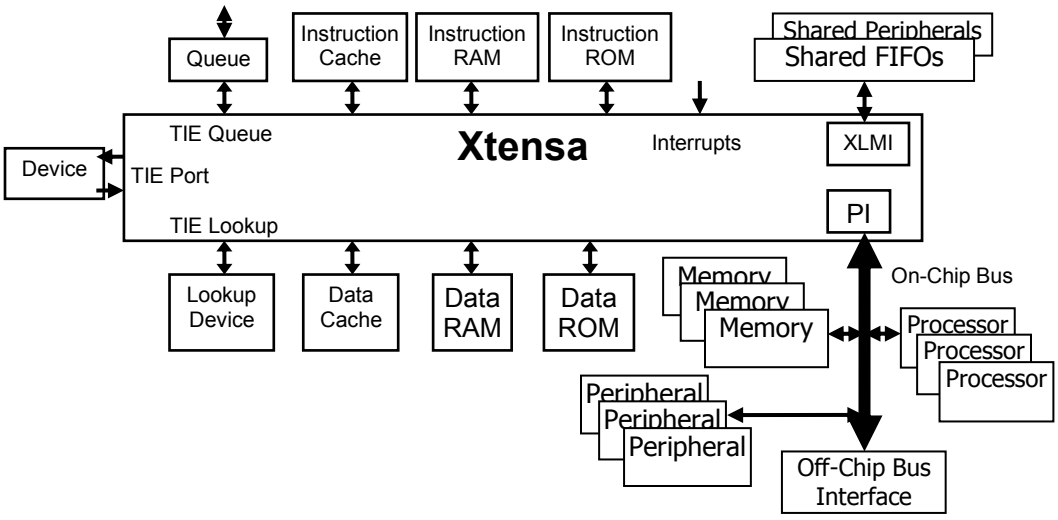

Figure 2. Tensilica Processor Interfaces

physical registers is 32 or 64[1]. Users can select pre-defined options such as a floating-point co-processor (FPU) and can define custom instruction set extensions using the Tensilica Instruction Extension language (TIE) [49]. The TIE compiler generates a customized processor, taking care of low-level implementation details such as pipeline interlocks, operand bypass logic, and instruction decoding. In addition, the Tensilica system generates a complete verification environment for the processor, including a standalone testbench with processor state monitors and more than 400 diagnostic test programs for the ISA and all processor features.

Using the TIE language, designers can add registers, register files, and new instructions to improve performance of the most critical parts of the application. Multiple operation instruction formats can be defined using the Flexible Length Instruction eXtension (FLIX) feature to further improve performance [19]. Another feature of the TIE language is the ability to add user-defined processor interfaces such as simple input or output wires, queues with buffers, and lookup device ports. These interfaces can be used to interconnect multiple processors or to connect a processor to other hardware units. The base Xtensa pipeline is either five or seven stages and has a user selectable memory access latency of one or two cycles. A two-cycle memory latency allows designers to achieve faster clock cycles or to relax timing constraints on memories and wires.

Tensilica provides many options for memory and external device interfaces (Figure 2): instruction and data caches, instruction and data RAMs and ROMs, shared memories, queues and FIFOs.

For Smart Memories we used many pre-defined Tensilica options: 32-bit integer multiplier and divider, 32-bit floating point unit, 64-bit floating point accelerator, seven stage pipeline with 2-cycle memory access and 4 scratch registers used for the runtime. Furthermore, we added several FLIX instruction formats that can execute up to 3 instructions per cycle. We also used several features that help with software debugging: the On-Chip Debug (OCD) option enables a GDB debugger running on a host PC to connect to the Tensilica processor via the JTAG interface, and an instruction trace port option, which can record an instruction trace to local memory.

We used Tensilica tools extensively throughout the Smart Memories project. First, we developed an architectural simulator using a generated processor model and Tensilica's Xtensa Modeling Protocol (XTMP) to interface to the memory system model [47]. Then, we used the Tensilica Processor Generator to produce synthesizable processor RTL, a processor standalone testbench, and a set of diagnostic test programs for the processor. This greatly reduced the amount of design and verification work. During the project we found an additional advantage of using IP that someone else was developing – it got better with time. When we started the project, the FLIX capability was not released, and the simulation system did not have a very fast, direct execution mode. Both were developed during the project and used in our design. Eventually Tensilica even added coherent cache support, but it was too late to use in our design.

5. SMART MEMORIES, VERSION II

Although using third-party IP blocks such as Tensilica processors is a very attractive idea for both industrial system-on-chip (SOC) designs and computer architecture research prototypes, in practice IP blocks such as processors are often challenging to use [37]. This section discusses the challenges and the solutions for interfacing a Tensilica processor to the rest of the Smart Memories design. To understand the minimum properties required from the processing core, we first turn to review some of the details of the memory system of Smart Memories.

When the project was restarted using Tensilica cores, the basic memory system design remained relatively unchanged. As shown in Figure 3, all on-chip memories consist of runtime-configurable memory mats. In each tile, the memory mats are interconnected through an Inter-Mat Communication Network (IMCN) (Figure 3b)—a fast path for exchanging memory control and state information to implement composite storage structures such as instruction and data caches (Figure 3c). In this example, the mats combine to form a two-way cache: mats 6 and 11 are used as tag storage while mats 7-10 and 12-15 store the data. The memory configuration consists of two parts. The first is configuration state that is located throughout the memory system. This state determines the function of each mat and various routing

[1]Later, 16-register option without windows also became available.

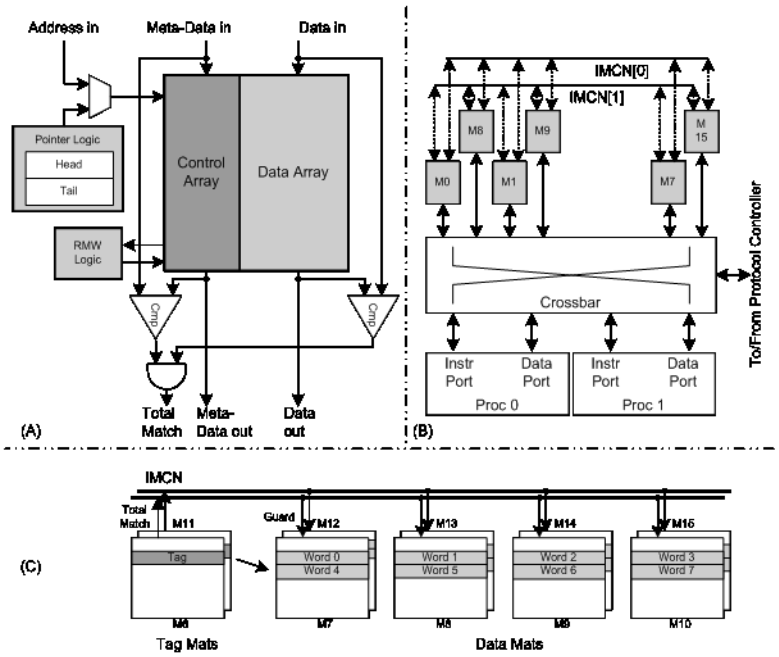

Figure 3. Tile memory organization: a) bock diagram of the memory mat; b) tile crossbar and IMCN; c) example cache configuration

connections, e.g. which mats contain tags and which mats contain data, whether the result of IMCN messages should affect the outcome of a given operation, etc. Since we want a single application to have access to a number of different types of memory operations, we also associate an opcode with each memory access. These opcodes are the second part of the memory configuration. They select which configuration bits are associated with the memory operation, how the metadata bits should be updated, and how the metadata and IMCN results should affect the memory access. This allows us to use the metadata bits associated with the data to implement full/empty bits and provide different types of load and store instructions to implement memory operations that support per-word synchronization, as well as track the coherence state of the cache lines and implement an LRU policy for the tag mats. In a transactional memory mode, opcodes are used to indicate speculative read and write operations which update metadata bits to indicate the transaction's read and write sets.

Having head/tail pointers associated with each of the memory mats makes it easy to efficiently implement hardware FIFOs. These are sometimes used to augment cache structures, e.g. to store the addresses of a transaction's write set, to be used later at commit time [16]. In streaming mode, the memory mats are aggregated into addressable scratchpads. FIFOs are also important for streams since they can be used to capture producer-consumer locality between processors.

While there is great flexibility in the memory system and associated protocol controller, the processor's task is quite simple. It needs to the ablility to generate load/store instructions with different opcodes—to send semantically different operations to the memory system—and the ability to tolerate occasional delays in getting the requested data back. With different "colored" memory access types, we can interpret these opcodes as FIFO load/store operations, synchronized load/store operations, gang operations (e.g., invalidating a transaction read/write set) or even simple metadata write/compare/read operations as needed for a given memory model. For the Smart Memories architecture, switching to a Tensilica processor solution would be feasible if it could support different types of memory operations and the special architectural semantics that they imply, and be able to recover from speculative execution that needed to be nullified.

There were two main issues that we needed to resolve to use the Tensilica processors in our architecture. The first was how to connect it to our coherent memory system, and the second was how to make the processor recover from mis-speculation.

5.1 Interfacing the Tensilica Processor to Smart Memories

Although Tensilica provides many options for memory interfaces (Figure 2), none of these interfaces can be used directly to connect the Tensilica processor to the rest of the Smart Memories system. The Xtensa processor has interfaces to implement instruction and data caches, but these interfaces do not support coherence operations, and thus could not be used for the multiprocessor Smart Memories architecture. The Xtensa cache interfaces connect directly to SRAM arrays for cache tags and data, and the cache management logic is fixed. As a result, it is impossible to modify the functionality of the Xtensa caches or to re-use the same SRAM arrays for different memory structures like local scratchpads.

To make the problem even more difficult, in addition to simple load and store instructions, the Smart Memories architecture must support several other memory operations that utilize the extra metadata bits. While these memory operations can easily be added to the instruction set of the processor using the TIE

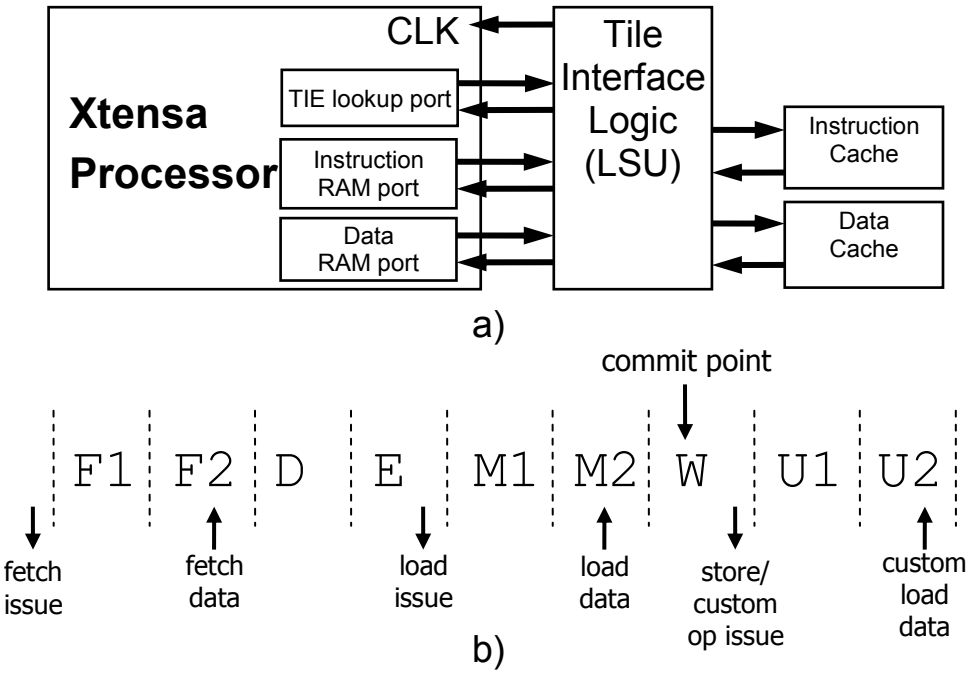

Figure 4. a) Interfacing Tensilica processor to Smart Memories; b) processor pipeline

language it is impossible to extend Xtensa interfaces to natively support such instructions.

To resolve these issues, we decided to use instruction and data RAM interfaces as shown in Figure 4a, instead of cache interfaces. Rather than connecting these interfaces directly to the memory arrays, we route all instruction fetches, loads and stores to the tile interface logic (Load Store Unit), which converts them into actual control signals for memory blocks used in the current configuration. Special memory operations are sent to the interface logic through the TIE lookup port, which has the same latency as the memory interfaces. If the data for a processor access is ready in 2 cycles, the interface logic sends it to the appropriate processor pins. If the reply data is not ready (e.g. due to cache miss, arbitration conflict in tile crossbar or remote memory access), the interface logic stalls the processor clock until the data is ready.

The advantage of this approach is that the instruction and data RAM interfaces are very simple: they consist of enable, write enable/byte enables, address and write data outputs and return data input. The meaning of the TIE port pins is defined by instruction semantics described in TIE. Processor logic on the critical path is minimal. The interface logic is free to perform any transformations with the virtual address supplied by the processor. Moreover, the semantic interpretation of the TIE port accesses is determined independently and can even be runtime configurable.

Adding special memory instructions to the architecture does add one complication. Special load instructions can modify metadata bits, i.e. they can alter the architectural state of the memory. Standard load instructions do not have side effects, i.e. they do not alter the architectural state of the memory system, and therefore they can be executed by the processor as many times as necessary. Loads might be reissued, for example, because of processor exceptions as shown in Figure 4b: loads are issued to the memory system at the end of the E stage, load data is returned to the processor at the end of the M2 stage, while the processor commit point is in the W stage, i.e. all processor exceptions are resolved only in the W stage. Such resolution may ultimately result in re-execution of the load instruction. Stores, by contrast, are issued only in the W stage after the commit point.

Therefore, because it would be very difficult to undo any side effects of special memory operations, they are also issued after the commit point in W stage, and the processor pipeline was extended by 2 stages (U1 and U2 in Figure 4b) to have the same 2-cycle latency for special load instructions.

However, having different issue stages for different memory operations creates the memory ordering problem illustrated in Figure 5a. A load following a special load in the application code is seen by the memory system before the special load because it is issued in the E stage. To prevent such re-ordering, we added pipeline interlocks between special memory operations and ordinary loads and stores. An example of such an interlock is shown in Figure 5b. The load is stalled in the D stage for 4 cycles to make sure the memory system sees it 1 cycle after the previous special load. One extra empty cycle is added between 2 consecutive operations to simplify memory system logic for the case of synchronization stalls. This does not degrade performance significantly because such combinations of a special load followed by a standard load are rare.

This addition of such an interlock alters the semantics of the core ISA of the Tensilica processor and is outside of the allowed range of user customizations. Yet we would not be able to implement our programmable memory system without it. Fortunately, we were able to work closely with Tensilica's engineering team to enable this change and get workarounds for the compiler issues that this change caused.

Another issue related to the interface between the Tensilica processor and the memory system is timing. Processor designers usually avoid stalling the processor clock on cache miss because it

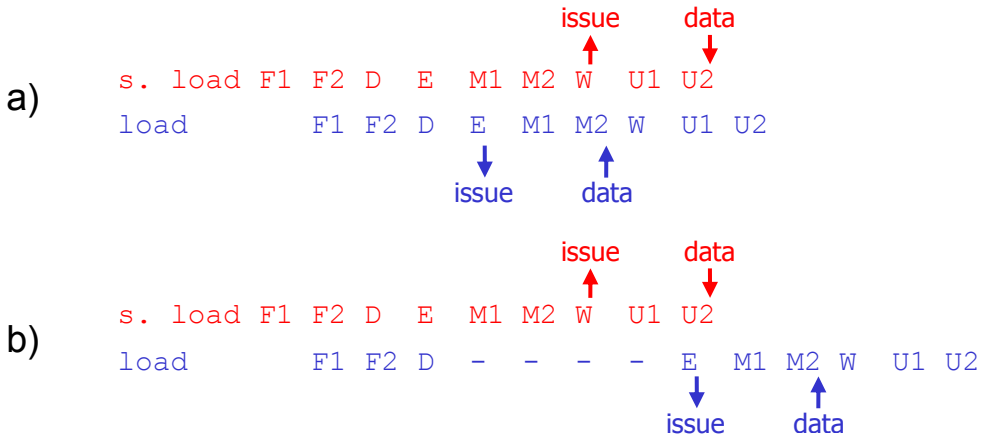

Figure 5. Ordering of memory operations: a) without interlock, b) with interlock

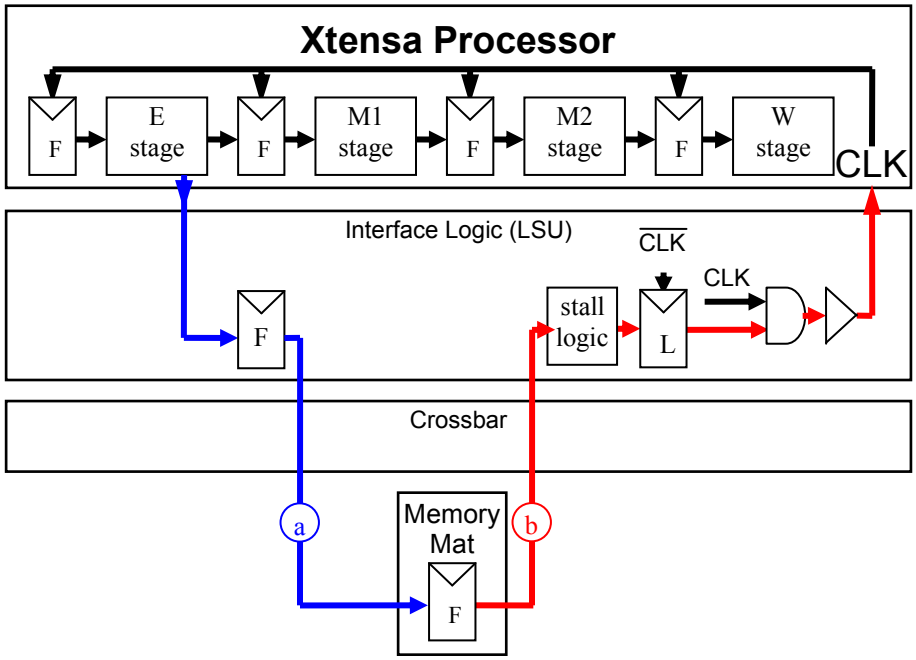

Figure 6. Tile timing critical paths: a) forward path from processor to local memory/cache; b) reverse path from local memory/cache to processor

will always create a critical path. The clock gate signal is needed very early to compensate for the clock tree delay to avoid glitching the clock. Instead modern machines disable state updates for squashed instructions, i.e. the write-back stage of the instruction that caused the cache miss. Unfortunately for us, the processor is a black box—we cannot change its internal operation (except by adding features using TIE). Since we were converting an SRAM interface into a cache interface, the only way to handle cache misses was to stall the processor clock. We stall the processor clock in the case of cache miss, crossbar arbitration failure or off-tile memory mat access (Figure 6). While this is a clean functional solution, it does create a tight timing path.

The tile timing is therefore determined by very tight timing constraints on the processor clock signal as shown in Figure 6. The forward path for the memory operation data issued by the processor goes through the flop in the interface logic and then through the flop in the memory mat. In the reverse path, the output of the memory mat goes to the stall logic and determines whether the processor clock should be stalled or not. To avoid glitches on the processor clock, the output of the stall logic must go through a flop or latch clocked with an inverted clock. The whole reverse path including memory mat, crossbar and stall logic delays must fit in a half clock cycle. This half cycle path is the most critical in the design and determines the clock cycle time. To relax timing constraints, the processor is clocked with an inverted clock: the delay of the reverse path must fit within the whole clock cycle, rather than just the half cycle. This shift does shorten the time for the E stage to generate the results, but since the processor does not limit the clock of this system, this path is fast enough.

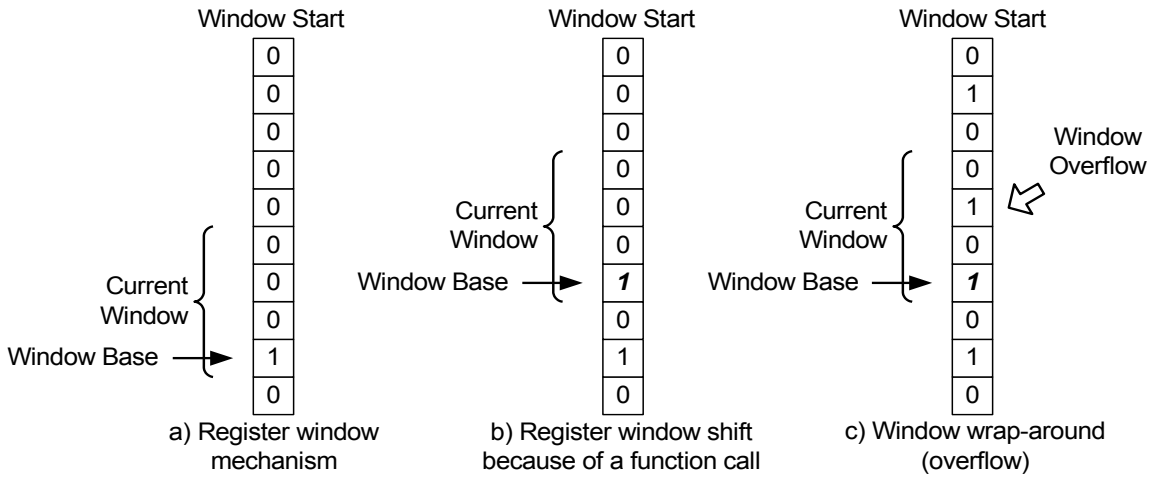

Figure 7. Register windows

5.2 Using a Register Windows Mechanism for Processor State Check-pointing

We leverage another feature of the Tensilica architecture to aid implementation of modes that support speculative execution. To checkpoint processor state at the beginning of speculative execution, previously proposed thread-level speculation (TLS) architectures modify the register renaming mechanism in out-of-order execution processors [9][43] or utilize a shadow register file [15]. It is possible to accomplish the needed checkpoint with little overhead and almost no hardware support (no shadow registers) in a machine with register windows such as Tensilica. In order to explain the proposed approach, let's consider an example of a windowed register file that consists of 64 physical registers, divided into groups of four, with a 16-register window being visible at any time. Figure 7a shows the two registers that control the window mechanism. Window Start has one bit per group of four registers, and indicates where a register window starts. Window Base is a pointer to the beginning of the current window in Window Start.

On each function call, as directed by the compiler, the register window shifts by 4, 8 or 12 registers (Figure 7b). Register window overflow occurs when, after shifting, Window Start has 1 within the current window (Figure 7c). In this case, an exception handler is invoked to spill the registers over to the memory.

When the processor runs a speculative thread, register values can fall into one of the following categories:

- constants, which are passed to the speculative thread and are not changed during execution;
- shared variables, which are modified by other threads and must be read from memory before they are used for computation;
- temporary values, which are alive only during the execution of this thread;
- privatized variables, such as loop induction variable or reduction variables.

The first three categories do not need to be saved at the start of a speculative thread, since they are not changed at all or are reloaded or recalculated. To simplify the violation recovery process, we have forced privatized variables to go through memory as well: the values are loaded at the start of the speculation and are saved back to memory at the end. Software overhead is typically quite small because speculative threads usually have few privatized variables.

If a speculative thread does not contain any function calls, the register window will not move during the execution of the speculative thread. As discussed, since the registers in the active window do not change, the recovery process after mis-speculation is very rapid because no restoration of the register values is required. However, if there is a function call in the speculative loop body, the register window will be shifted by the function call. If a violation is detected while the thread is in the middle of the function call, the state of the register window must be recovered correctly. For this purpose, two instructions and a special register are added to the processor for saving and restoring Window Start and Window Base values atomically. In order to keep the recovery operation simple and fast, the exception handler for the window overflow is also modified to avoid spilling the registers when the execution is speculative. This way, it is not necessary to read back the spilled values into the register file in the case of violation; the window exception handler is simply stalled until the thread becomes non-speculative and can commit its changes to the memory system.

In comparison with a shadow register file approach, our technique requires little extra hardware: a special register to save values of Window Start and Window Base, and two new instructions. In comparison with a purely software approach (which takes tens to a hundred cycles), our technique is significantly faster: it requires one instruction to save Window Start and Window Base and only a few store instructions for privatized variables, since a typical speculative thread rarely has more than two privatized variables. It should be noted that this checkpointing technique is not applicable to non-windowed architectures such as MIPS, because function calls may overwrite any register regardless of how it was used in the caller function.

5.3 Software and Benchmarking Issues

In theory developing a custom processing core that exactly fits the architectural requirement of the system will provide better performance and better power efficiency than reusing an off-the-shelf core. However, in practice the better software development infrastructure that accompanies an existing core can easily allow the programmer to achieve higher application performance. The

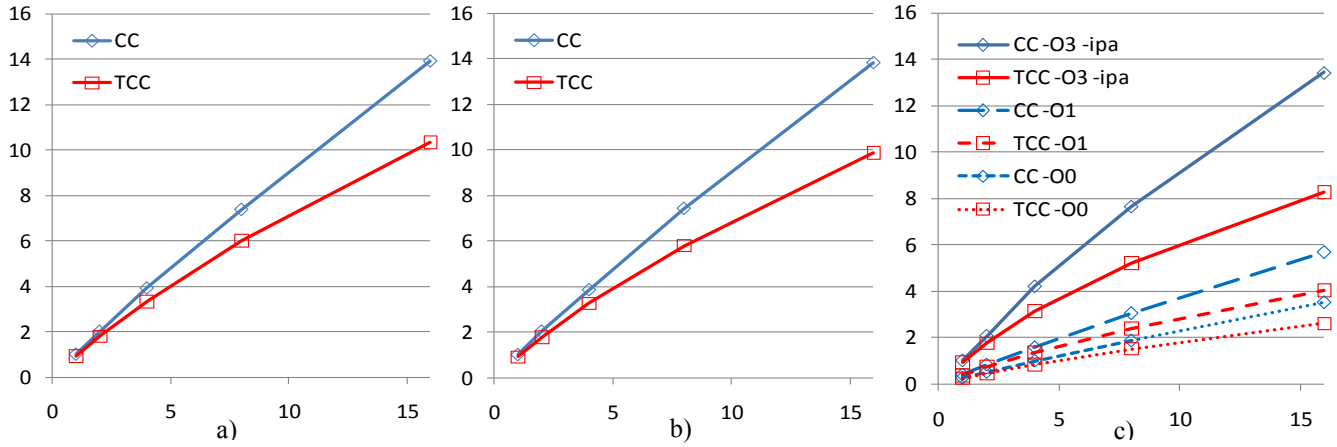

Figure 8. Speedups of TCC and cache-coherent (CC) versions of barnes: a) compiled with –O0, b) compiled with –O1, c) compiled with –O3 –ipa vs. a) and b)

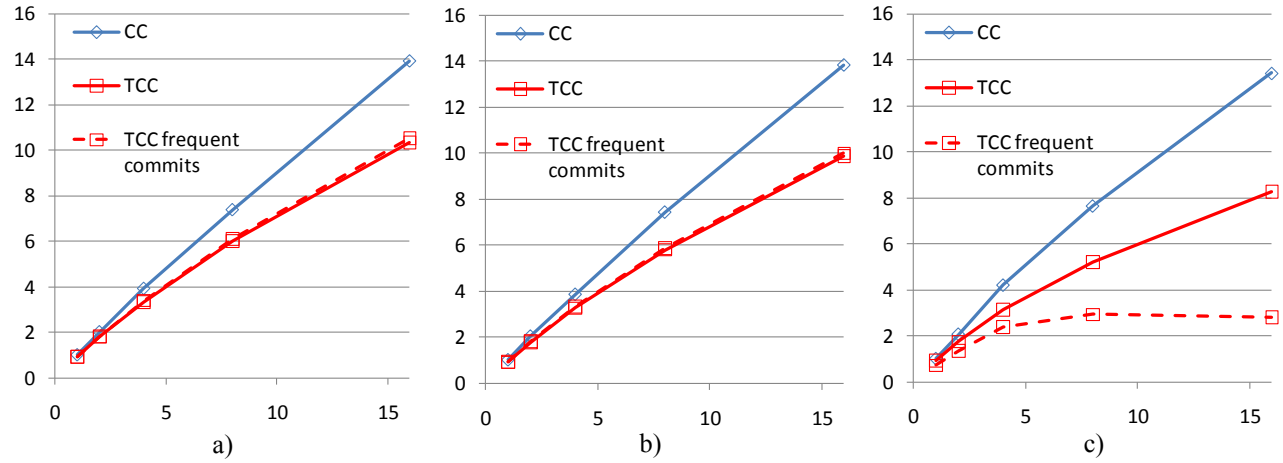

Figure 9. TCC barnes optimization: a) complied with –O0, b) compiled with –O1, c) compiled with –O3 –ipa

overall system performance is as dependent on the software tools quality (compiler, linker, etc) as it is on the hardware.

The availability of a high-quality optimizing Tensilica compiler makes benchmarking of the architecture simpler and more credible. We did not have to resort to hand-coding of microbenchmarks [13], instead we could concentrate on performance evaluation using applications such as SPLASH-2 [50], and the TCC version of SPLASH-2 benchmarks [31]. Not surprisingly the level of compiler optimizations affected benchmark performance and the way performance scaled with number of processors.

Figure 8 shows speedups versus the number of processors for two versions of the same barnes application. One of them is the original code from the SPLASH-2 suite designed for shared memory cache-coherent architectures (designated as CC), the other is a version of the same application converted for a transactional memory architecture, TCC [31]. Figure 8a shows performance scaling for un-optimized executables, compiled with option –O0, and normalized to a shared memory version running on a single processor. Similarly, Figure 8b shows performance scaling for executables compiled with option –O1. Finally, Figure 8c shows performance scaling for executables with highest level of optimization, compiled with options –O3 and –ipa (ipa stands for inter-procedural analysis).

The version compiled with the highest level of optimization is approximately 4 times faster than the un-optimized version, and 2.5 times faster than the version compiled with option –O1. Also, at the highest level of optimization performance, the scaling of the TCC version is noticeably worse: on a 16-processor configuration it is 1.62x slower than the shared memory version, while in the case of un-optimized code (Figure 8a) it is only 1.35x slower than the shared memory version. Of course, these results are dependent on the cache configuration and details of our implementation of TCC, which uses software in the runtime to implement part of the protocol [41]. In fact most of this overhead comes from executing extra instructions during transaction commit. This overhead is less than 10% on a single processor configuration for all cases (Figure 8).

We spent significant effort trying to optimize and tune the performance of the TCC version of barnes. Figure 9 shows performance scaling of the tuned version of barnes (TCC) versus the original version from [31] (designated as **TCC frequent commits**). In simulating this application, it became clear that the original TCC version did not scale well in the most optimized case (Figure 9c) because of the large number of commits inside

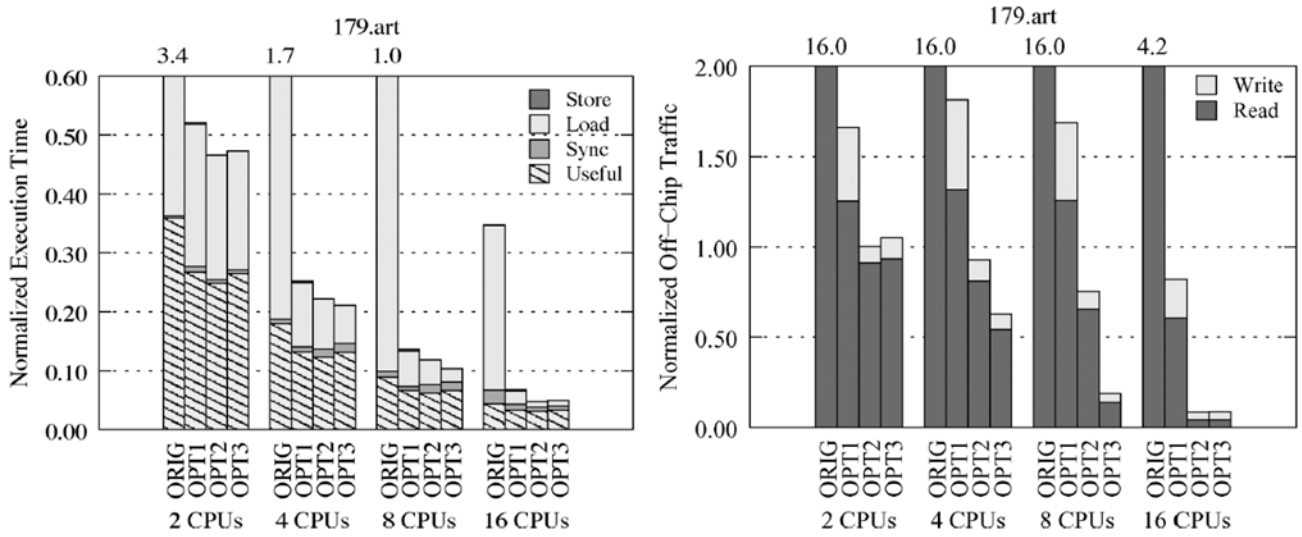

Figure 10. The effect of stream programming optimizations on the shared memory 179.art's performance and off-chip traffic [27]

recursive function calls of the main computation loop. Elimination of these frequent unnecessary commits improved performance by approximately 3x.

Similarly, when we tried to compare streaming and shared memory architectures using the Smart Memories infrastructure [27] we found that the result of comparison is highly dependent on the software optimizations of the benchmarks. In fact, many streaming optimizations can be applied to a shared memory version of the same benchmark with significant improvements in performance and energy dissipation. Figure 10 shows normalized execution time and off-chip memory traffic for a parallelized version of 179.art benchmark from the SPEC2000 suite. ORIG is the original version; OPT1, OPT2 and OPT3 are successively more optimized versions of the same applications (a detailed description of the optimizations can be found in [27][41]). While the original shared memory version of 179.art is significantly slower than streaming, the most optimized shared memory version has approximately the same performance. Enabling this type of software tuning completely changed the conclusion about the importance of creating a streaming memory system.

5.4 Status and Lessons Learned

Clearly, no matter how much effort Tensilica or any other vendor put in to it, no off-the-shelf processing core would ever hold the exact features required for a new architecture. However, with the current generation of extensible processors we can come very close to that goal. With the ability to add new instructions and ports (using TIE and TIE ports in the Tensilica system) and being willing to accept practical solutions with a few workarounds, new architectures can be realized and can significantly benefit from software tooling reuse. Especially important are compiler optimizations, as those enable true comparisons of the new architecture to well established ones.

Leveraging Tensilica cores and software stack we implemented an extensible multi-processor architecture. Both the system simulator and the RTL design were tested to run with up to 32 active processors (four quads). Eventually, the architecture was validated against three specific models: cache-coherent shared memory [1], streaming [45] [21] and TCC [16].

An 8-core (one quad) test chip was fabricated using ST Microelectronics 90 nm technology. It contains four tiles, each with two Tensilica processors, and a shared protocol controller. The total chip area is 60.5 mm^2 and contains 55 million transistors. The chip successfully passed extensive testing, starting from JTAG configuration, on-chip memory read-write tests, and running scaled-up application programs in the three target modes of operation. We are currently working on bringing up 32-processor system with 4 Smart Memories test chips.

6. CONCLUSIONS

A pre-existing extensible processor system, such as that provided by Tensilica, can turn an otherwise unmanageable, overly-ambitious architecture project into a feasible effort successfully completed despite severely limited manpower resources.

For instance, building a successful computing system at a university has always been a difficult task. The most successful projects manage the amount of work required by focusing their effort on a few core research issues, and try to leverage existing tools and techniques from others for the remaining non-core issues. This issue of focus has become especially acute in building new processor systems, where the level of software infrastructure that is needed to create a useable platform is quite large. It is for this reason that much of the work on multiprocessors has used existing ISAs and even processor boards, and why so many new architecture demonstrations are based on the open source SPARC designs. Of course, the downside of using an established ISA/architecture is that the new ideas become more difficult to realize.

When working on the reconfigurable Smart Memories project, we used an extensible processor system that could produce a customized processor and necessary software tools. While this decision constrained many aspects of the overall system, it allowed us to focus on the design of the reconfigurable memory system and to successfully design and fabricate a test chip with a production quality software environment. The strong software tools made it easier to run more applications, but this just drove home the point that the evaluation of new architecture is tricky. Since the interaction between application sharing patterns and the

underlying hardware capabilities can be surprising, it requires a significant amount of work on application optimization and tuning to get meaningful results.

Our current work continues to explore the potential of extensible processors. We have found that Smart Memories can provide the basis for evaluating a number of architectural ideas, since both processor and memory system are highly flexible. This has led to our current research in building a CMP generator [42]. The idea is to use the reconfigurable Smart Memories architecture as a virtual prototyping platform that lets an application designer configure memory operations, protocols, and the underlying processor instructions, to tune the machine for a specific application or application class, while still maintaining a high-quality software development environment. Preliminary results indicate that this approach could create computing solutions that are orders of magnitude more energy efficient than conventional approaches.

7. ACKNOWLEDGMENTS

This work would have not been possible without great support and cooperation from many people at Tensilica: Chris Rowen, Dror Maydan, Bill Huffman, Nenad Nedeljkovic, David Heine, Govind Kamat and others. The authors also would like to acknowledge support from DARPA and ST Microelectronics. The authors also would like to thank Han Chen, Kyle Kelley, Francois Labonte, Jacob Chang and Don Stark for their help and support.

8. REFERENCES

[1] S.V. Adve, K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. Computer, vol.29, no.12, pp. 66-76, December 1996.

[2] A. Agarwal, et al. Sparcle: An Evolutionary Processor Design for Large-Scale Multiprocessors. IEEE Micro Magazine, vol. 13 no. 3, pp. 48-61, May 1993.

[3] J. H. Ahn, W. J. Dally, B. Khailany, U. J. Kapasi, A. Das. Evaluating the Imagine Stream Architecture. Proceedings of the International Symposium on Computer Architecture, June 2004.

[4] K. Arvind, R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. IEEE Transactions Computers, 39, 3, pp. 300-318, Mar. 1990.

[5] W. Baek, C. C. Minh, M. Trautmann, C. Kozyrakis, and K. Olukotun. The OpenTM Transactional Application Programming Interface. Proceedings of the International Conference on Parallel Architecture and Compilation Techniques, pp. 376-387, September 2007.

[6] R. Barua, W. Lee, S. Amarasinghe, A. Agarwal. Compiler Support for Scalable and Efficient Memory Systems. IEEE Transactions on Computers, November 2001.

[7] S. Borkar, R. Cohn, G. Cox, T. Gross, H. T. Kung, Monica Lam, Margie Levine, Brian Moore, Wire Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. Supporting Systolic and Memory Communication in iWarp. Proceedings of the International Symposium on Computer Architecture, Seattle, Washington, pp. 70 - 81, May 1990.

[8] F. C. Chow, J. L. Hennessy and L. B. Weber. The MIPS Compiler. Chapter 20 in "Optimization in Compilers", edited by F. Allen, B. K. Rosen, and K. Zadeck, by The Association for Computing Machinery, Inc., 1991.

[9] M. Cintra, et al. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors. Proceedings of the International Symposium on Computer Architecture, June 2000.

[10] A. Das, W. J. Dally, P. Mattson. Compiling for Stream Processing. Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 2006.

[11] Dehnert, J. C. and Towle, R. A. 1993. Compiling for the Cydra 5. Journal of Supercomputing, 7, 1-2, pp. 181-227, May 1993.

[12] A. Firoozshahian, A. Solomatnikov, O. Shacham, S. Richardson, C. Kozyrakis, M. Horowitz. A Memory System Design Framework: Creating Smart Memories. Proceeding of the International Symposium on Computer Architecture, pp. 406-417, June 2009.

[13] M. Gebhart, B. A. Maher, K. E. Coons, J. Diamond, P. Gratz, M. Marino, N. Ranganathan, B. Robatmili, A. Smith, J. Burrill, S. W. Keckler, D. Burger, and K. S. McKinley. An Evaluation of the TRIPS Computer System. Proceeding of the International Conference on Architectural Support for Programming Languages and Operating Systems, March 2009.

[14] R. Gonzalez. Xtensa: A Configurable and Extensible Processor. IEEE Micro, 20(2):60–70, Mar/Apr 2000.

[15] L. Hammond, B. Hubbert, M. Siu, M. Prabhu , M. Chen , K. Olukotun. The Stanford Hydra CMP. IEEE Micro, March-April 2000.

[16] L. Hammond, V. Wong, M. Chen, B. Hertzberg, B. D. Carlstrom, J. D. Davis, M. K. Prabhu, H. Wijaya, C. Kozyrakis, K. Olukotun. Transactional Memory Coherence and Consistency. Proceedings of the International Symposium on Computer Architecture, June 2004.

[17] M. Herlihy and J. E. Moss, "Transactional memory: architectural support for lock-free data structures," Proceedings of the International Symposium on Computer Architecture, pp. 289-300, May 1993.

[18] W. W. Hwu, R. E. Hank, D. M. Gallagher, S. A. Mahlke, D. M. Lavery, G. E. Haab, J. C. Gyllenhaal, and D. I. August. Compiler Technology for Future Microprocessors. Proceedings of the IEEE, vol. 83, no. 12, pp. 1625-1640, December 1995.

[19] D Jani, G Ezer, J Kim. Long words and wide ports: Reinventing the Configurable Processor. Hot Chips, August 2004.

[20] H. Kannan, M. Dalton, C. Kozyrakis. Raksha: A Flexible Architecture for Software Security. Hot Chips, August 2007.

[21] B. Khailany, W. J. Dally, S. Rixner, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, and A. Chang. Imagine: Media Processing with Streams. IEEE Micro, pages 35–46, Mar/Apr 2001.

[22] R. Krishnaiyer; D. Kulkami; D. Laven; L. Wei, C.-C. Lim; J. Ng; D. Sehr. An Advanced Optimizer for the IA-64 Architecture. IEEE Micro, vol.20, no.6, pp. 60-68, Nov/Dec 2000.

[23] R.C. Kunz. Performance Bottlenecks On Large-Scale Shared-Memory Multiprocessors. PhD Thesis, Stanford University, 2004.

[24] J. Kuskin, et al. The Stanford FLASH Multiprocessor. Proceedings of the International Symposium on Computer Architecture, pp. 302-313, April 1994.

[25] S. Leibson, J. Kim. Configurable Processors: A New Era in Chip Design. IEEE Computer, vol.38, no.7, pp. 51-59, July 2005.

[26] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. Proceedings of the International Symposium on Computer Architecture, pp. 148-159, 1990.

[27] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, C. Kozyrakis. Comparative Evaluation of Memory Models for Chip Multiprocessors. ACM Transactions on Architecture and Code Optimization, November 2008.

[28] G. P. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg. The Multiflow Trace Scheduling Compiler. The Journal of Supercomputing, vol. 7, no. 1-2, pp. 51-142, 1993.

[29] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, M. Horowitz. Smart Memories: a Modular Reconfigurable Architecture. Proceedings of the International Symposium on Computer Architecture, pp. 161–171, 2000.

[30] K. Mai, R. Ho, E. Alon, D. Liu, Y. Kim, D. Patil, and M. Horowitz. Architecture and circuit techniques for a 1.1-GHz 16-kb reconfigurable memory in 0.18u CMOS. IEEE Journal of Solid-State Circuits, 40(1):261–275, 2005.

[31] A. McDonald, J. Chung, H. Chafi, C. C. Minh, B. D. Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun. Characterization of TCC on Chip-Multiprocessors. Proceedings of the International Conference on Parallel Architecture and Compilation Techniques, pp. 63-74, September 2005.

[32] A. McDonald, J. Chung, B. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural Semantics for Practical Transactional Memory. Proceedings of the International Symposium on Computer Architecture, June 2006.

[33] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford Transactional Applications for Multi-Processing. Proceedings of the IEEE International Symposium on Workload Characterization, pp. 35-46, 2008.

[34] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems, pp. 48–60, May 1993.

[35] M. D. Noakes, D. A. Wallach and W. J. Dally. The J-Machine Multicomputer: An Architectural Evaluation. Proceedings of the International Symposium on Computer Architecture, pp. 224 – 235, 1993.

[36] D. Patterson. Reduced Instruction Set Computers. Communications of the ACM, vol. 28, no.1, January 1985, pp. 9-21.

[37] A. Rappaport. Semiconductor Value in the Post Fabless Era. Talk at the Stanford EE Computer Systems Colloquium, January 2009, http://www.stanford.edu/class/ee380/Abstracts/090128.html

[38] O. Shacham, Z. Asgar, H. Chen, A. Firoozshahian, R. Hameed, C. Kozyrakis, W. Qadeer, S. Richardson, A. Solomatnikov, D. Stark, M. Wachs, M. Horowitz. Smart Memories Polymorphic Chip Multiprocessor. Proceedings of the Design Automation Conference, July 2009.

[39] M.S. Schlansker, B.R. Rau. EPIC: Explicitly Parallel Instruction Computing. IEEE Computer, vol.33, no.2, pp.37-45, Feb 2000.

[40] A. Smith, J. Burrill, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D.C. Burger, and K.S. McKinley. Compiling for EDGE Architectures. International Conference on Code Generation and Optimization (CGO), March, 2006.

[41] A. Solomatnikov. Polymorphic Chip Multiprocessor Architecture. PhD Thesis, Stanford University, 2008.

[42] A. Solomatnikov, A. Firoozshahian, W. Qadeer, O. Shacham, K. Kelley, Z. Asgar, M. Wachs, R. Hameed, and M. Horowitz. Chip Multi-Processor Generator. Proceedings of the Design Automation Conference, June 2007.

[43] J. G. Steffan, et al. A Scalable Approach to Thread-Level Speculation. Proceedings of International Symposium on Computer Architecture, June 2000.

[44] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, S.J. and Eggers. The WaveScalar architecture. ACM Transactions on Computer Systems, 25, 2, 4, May 2007.

[45] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffmann, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe and A. Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. IEEE Micro, Mar/Apr 2002.

[46] 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. Proceedings of International Symposium on Computer Architecture, June 2004.

[47] Xtensa XTMP and XTSC. Tensilica. http://www.tensilica.com/products/devtools/hw_dev/sw_xtmp_xtsc.htm

[48] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreaMIT: A Language for Streaming Applications. Proceedings of the International Conference on Compiler Construction, pp. 179-196, 2002.

[49] A. Wang, E. Killian, D. Maydan, and C. Rowen. Hardware/Software Instruction Set Configurability for System-on-Chip Processors. Proceedings of the Conference on Design Automation, 2001.

[50] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. Proceedings of the International Symposium on Computer Architecture, pp. 24-36, June 1995.