Multicore Performance Optimization Using Partner Cores

Eric Lau[*], Jason E Miller[*], Inseok Choi[**], Donald Yeung[**], Saman Amarasinghe[*], and Anant Agarwal[*]

[*] *MIT Computer Science and Artificial Intelligence Laboratory*
[**] *University of Maryland Dept. of Electrical and Computer Engineering*

Abstract

As the push for parallelism continues to increase the number of cores on a chip, system design has become incredibly complex; optimizing for performance and power efficiency is now nearly impossible for the application programmer. To assist the programmer, a variety of techniques for optimizing performance and power at runtime have been developed, but many employ the use of speculative threads or performance counters. These approaches result in stolen cycles, or the use of an extra core, and such expensive penalties can greatly reduce the potential gains.

At the same time that general purpose processors have grown larger and more complex, technologies for smaller embedded processors have pushed towards energy efficiency. In this paper, we combine the two and introduce the concept of Partner Cores: low-area, low-power cores paired with larger, faster compute cores. A partner core is tightly coupled to each main processing core, allowing it to perform various optimizations and functions that are impossible on a traditional chip multiprocessor. This paper demonstrates that optimization code running on a partner core can increase performance and provide a net improvement in power efficiency.

1 Introduction

As multicore chips scale to larger numbers of cores, systems are becoming increasingly complex and difficult to program. Parallel architectures expose more of the system resources to the software and ask programmers to manage them. In addition, with energy consumption now a primary system constraint, programmers are forced to optimize for both performance and energy; a task that's nearly impossible without knowing the exact hardware and environment in which an application will run. As machines scale to hundreds of cores, it will no longer be possible for the average programmer to understand and manage all of the constraints placed upon them.

One approach to reducing the burden placed on programmers is the use of *self-aware* systems. Self-aware systems (sometimes also known as *autonomic*, *adaptive*, *self-optimizing*, *etc.*) attempt to automatically monitor the system and dynamically optimize their behavior based on runtime conditions. This reduces the amount of optimization a programmer must perform and frees them from the need to anticipate all possible system configurations *a priori*. However, on modern multicores, self-aware software systems must share the same resources being used by the application. Many run as extra threads, requiring additional context switches or a dedicated core. Some even interrupt the application to collect statistics and perform adaptations. Both of these can increase application runtime and energy consumption thereby requiring larger gains for the self-aware technique to be worthwhile.

To help reduce these costs and even enable new types of self-aware systems, we propose a new hardware structure called a *Partner Core*. A partner core is essentially a small, low-power core that is tightly-integrated with a primary core, allowing it to observe and adjust the behavior of the primary (or *main*) core. The self-aware runtime code can then be off-loaded to the partner core, freeing the main core to concentrate on the application. Whereas the main cores in a system are optimized for performance, the partner cores are optimized for energy efficiency and size. This allows the self-aware algorithms to be run very cheaply, more easily producing a net positive effect. Since the self-aware algorithms generally do not require as much computation as the application, the lower performance is perfectly acceptable.

To this end, we describe potential use cases for partner cores in several examples of self-aware applications

This work was funded by the U.S. Government under the DARPA UHPC program. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

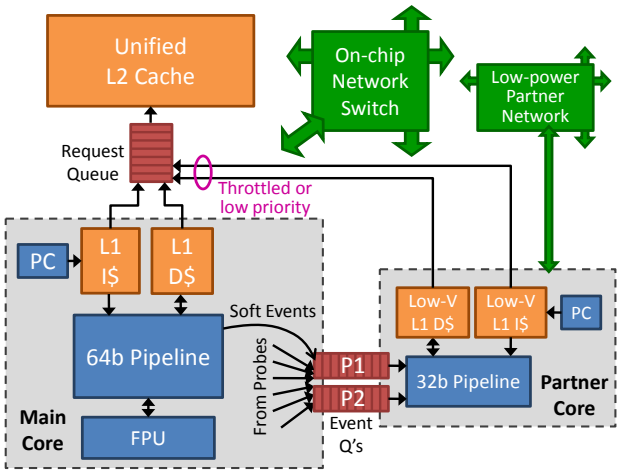

Figure 1: Architecture of multicore tile showing main and partner cores and the connections between them.

such as dynamic knobs for adaptive performance control, redundant multithreading for reliability, pointer tainting for security, and helper threads for memory prefetching.

We analyzed the performance of a processor using partner cores to run a memory prefetching *Helper Threads* [11]on the EM3D benchmark from the Olden suite [20]. Using partner cores and helper threads we are able to achieve application speedup of up to 3x and increase power efficiency by up to 2.2x.

2 Architecture

At a high-level, a partner core is simply a small, low-power core attached to a larger primary compute core. What makes it unique is its emphasis on low-power rather than performance and the way in which it is connected to the primary core. This section describes the architecture of a partner core, how it connects to the main core, and discusses some of the design considerations. Because we are targeting future multicores containing hundreds of cores, we assume a tiled architecture with on-chip mesh networks (similar to the Intel TeraScale prototype [3] or a Tilera TILE64 [22]) as a baseline. However, Partner Cores could also be used in other architectures and even benefit single-core chips.

2.1 Partner Core Design

Figure 1 shows the contents of a single tile in our proposed multicore architecture. The major components are the main compute core, the partner core, and the on-chip network switches. The main core contains a conventional 64-bit datapath with FPU plus L1 and L2 caches which miss to DRAM over the on-chip network.

The partner core uses a smaller, simpler pipeline to save area and energy. A narrower datapath is sufficient as long as the ISA allows it to manipulate the larger words of the main core when needed (probably taking multiple cycles). The partner core will probably not require an FPU, although it is possible that future self-aware algorithms might demand one. By reducing the size of the register file and number of pipeline stages, we can further reduce the complexity and footprint.

While the partner core contains its own L1 caches (or possibly a unified L1 cache), it shares the L2 cache with the main core. This allows the partner core to easily observe and manipulate the main core's memory. Partner core accesses can be throttled or given lower priority to reduce the chances that partner core accesses will negatively impact main core performance.

The most important features of the partner core are the observation and action interfaces with the main core. The tight coupling between the two cores is what enables the partner core to run self-aware algorithms efficiently. The partner core has the ability to directly inspect some of the main core's key state such as status registers, performance counters, L2 cache, etc. However, it would be extremely inefficient if the partner core had to poll for everything. Therefore, the main core is augmented with *event probes*, small pieces of hardware that watch certain state and generate events when specified conditions are met. Event probes are essentially performance counters with a programmable comparator and extra registers to store trigger values. They can be set to watch for specific values or values within or outside of a specified range.

When an event probe fires, an event is generated and placed in a queue between the cores (Figure 1). Events can also be generated directly from software, *e.g.,* to implement the Heartbeats API [8]. The optimal number and configuration of queues is still an open question; however, we expect to need at least two or three queues with different priority levels. When a queue fills, there are two options: either the main core must be stalled or events must be dropped. Some uses of Partner Cores may require reliable delivery of events while others may be best-effort optimizations. Therefore, both options should be available. Finally, we expect the partner core to periodically poll the queues and process events. However, certain events might need urgent handling and should generate an interrupt on the partner core. All of these options could be provided either through dynamic configuration or multiple queues with different properties.

2.2 Area

A key feature of the partner core is that it is purely speculative, meaning that its instructions are never committed in the main computation. This relaxes the speed re-

Core	Node (nm)	SRAM (bytes)	Size (mm^2)	Scaled Size (mm^2)
μcontroller	65	128K	1.5	0.05
RAW	180	128K	16	0.25
Intel Core 2	65	2M/4M	80	9.0

Table 1: Size specifications for selected core types; scaled sizes are for the 22nm node.

quirement, allowing us to design partner cores with minimal complexity and area, and without much concern for switching speed. Much of the recent work in small, low-power processors for embedded systems can be used for the design of a partner core. A good example is found in [14], where a 65nm ultra-low power 16-bit microcontroller contains core logic that occupies just $0.14mm^2$, and 128K of low-power SRAM that occupies $1.36mm^2$.

This is in contrast with modern general-purpose cores like the Intel Core 2, where logic alone covers 36 mm^2 of silicon [6]. Admittedly, a tiled multicore system will have cores that are considerably less complex. A better basis for comparison may be a tile in the RAW system, which contains sixteen single-issue, in-order RISC processors at 180nm technology [21].

Table 1 shows the sizes of these cores in their native technology, and also scaled to the 22nm node using first-order scaling trends. The microcontroller was adjusted to accommodate a 64-bit datapath, and a 16K SRAM. At this feature size, such a microcontroller would occupy around 20% of a RAW tile. Processors in future many-core systems will likely be more complex than the RAW processor, and so 10% tile area is a reasonable target for a partner core.

2.3 Power

One of the primary design metrics for massively parallel systems is energy per operation. Fortunately, with the aforementioned relaxed speed constraint, we can design the partner core for minimal energy consumption.

Voltage scaling has already emerged as one of leading low-power solutions for energy efficient applications, mainly due to the quadratic savings in power (eg. [10, 4]). As the technology for ultra-low voltage circuits becomes more mature, work has been done in the design of entire low-power systems-on-a-chip. Sub-threshold circuits have received particular attention: [23] implemented a 130nm processor with an 8-bit ALU, 32-bit accumulator, and a 2Kb SRAM functioning at 200mV; [14] demonstrated a 65nm processor with a 16-bit microcontroller and 128Kb of SRAM functioning down to 300mV. These implementations are likely too simple for a primary application core, but the energy efficiency of such microcontrollers are ideal for a partner core.

	μController	TilePro64	Intel Core 2
Energy	27.2pJ	286pJ	15.5nJ

Table 2: Energy per cycles for selected core types.

Table 2 shows the energy consumption per cycle of several cores [1, 7, 14], where it is clear the 16-bit microcontroller's power demands are a fraction of the full-featured cores. The Tilera TilePro64 tile contains a RISC processor that most closely resembles a core in a massively multicore system, and consumes 10x more energy per instruction than the microcontroller.

3 Potential Applications

As mentioned before, self-aware systems are an attractive approach for reducing the burden for programmers of increasingly complex parallel systems. However, continuous observation of an application is difficult on traditional single core processors: polling performance counters for a rare event is unrealistic, and interrupt-based schemes are expensive.

Partner cores can be used for introspection, decision-making, and adaptation, the key components of the ODA (Observe-Decide-Act) loop that is characteristic of these systems. Introspection includes the observation of main core state such as status registers, performance counters, cache contents, as well as application status through something like the Heartbeat API [8]. A variety of decision algorithms can be used including those based on control theory, machine learning, and simple heuristics. Finally, partner cores can take action by modifying hardware or application contents through its tight coupling with the main core. Along with the favorable energy efficiency of a partner core, this opens the door for many applications.

3.1 Dynamic Knobs

The authors of [9] describe PowerDial, a system for dynamically adapting application behavior to fluctuations in load and power. PowerDial is provided with a set of inputs, parameters, and QoS (Quality of Service) metrics along with the application. Using these, PowerDial computes a set of control variables which it can then use to dynamically move the operation of the application to different parts of the performance/QoS solution space.

PowerDial then implements a feedback-based runtime control mechanism with a set of "dynamic knobs" based on these control variables. The system listens for heartbeats that are emitted by the application [8] and uses control theory to compute a setting for its dynamic knobs at each heartbeat. One drawback of PowerDial is that the runtime system executes in the same context as the

application thread. Therefore if the application generates heartbeats too frequently, the PowerDial overhead can overwhelm the application. Placing the runtime system on a partner core alleviates this problem by isolating and constraining the resources that it can consume. Further, since the partner core is tightly coupled, heartbeats can be generated and processed more quickly, allowing for faster adaptation. For example, in the x624 video encoding application described in the paper, the application emits a heartbeat for each rendered frame. This means that PowerDial can only react and retain QoS across frames, which in practice, might cause frames to skip. With partner cores, heartbeats could be instrumented at the macro-block level, allowing PowerDial to smooth out performance within a single frame.

3.2 Reliability

Another pressing issue with the reduction of supply voltages is the increased vulnerability to transient faults. One fault-detection approach for microprocessors is redundant multithreading (RMT), which simultaneously runs two identical threads and compares the results [19]. This can be implemented on a standard SMT processor, where the trailing thread waits for confirmation with the leading thread before committing any results. This approach possesses the performance drawbacks intrinsic to sharing the same context. Chip-level redundant multithreading (CRT) executes a loosely synchronized redundant thread on a separate processor. However, this relies on the assumption that the distance between cores is short [19]. This has obvious scalability issues for massively multicore systems, especially with the emergence of wire delay being the dominant contributor to latency.

Partner cores are ideal for redundant multithreading. It runs redundant threads on separate hardware, so it avoids the slowdown that affects SMT implementations, and gains the advantages of a CRT implementation. It also overcomes the issue with CRT since partner cores are paired with each compute core, and so are easily scalable to tiled multicore architectures. Finally, the physical proximity of the partner core provides it access to the compute core's pipeline, providing access to the branch history table and branch table buffer, allowing even more opportunity for performance optimization.

3.3 Security

Except perhaps for some specific applications, a partner core has no need to run application-level code. Even in such rare cases (*e.g.,* execution of a leading thread for RMT), the system has full control over what is run on the partner core. This keeps the partner core decoupled from potentially malicious code running on the main cores, allowing it to monitor the health of the system at a separate layer from the application.

As an example, the partner core can monitor the memory accesses in the main core, flagging illegal accesses to kernel memory, or any other space not belonging to the application. Such a security feature could incorporate pointer tainting. Pointer tainting is a form of dynamic information flow tracking that marks the origin of data by a taint bit inaccessible by the application (*e.g.,* in a partner core). Tracking the propagation of the tainted data, a secure system can determine whether any data derived from the tainted origin is stored in an illegal location. While this is implementable in software, hardware implementations of pointer tainting accumulate minimal overhead [5].

4 Case Study: Memory Prefetching

One of the main bottlenecks on performance is memory latency. While clock frequencies of modern processors have advanced rapidly, memory access times have remained remarkably slow in relation. Memory prefetching at the thread level attempts to hide cache miss latency by triggering problematic loads in a separate *helper thread*. This section studies the potential benefits of running helper threads on partner cores.

4.1 Methodology

We instrumented helper threads for EM3D, a benchmark from the Olden suite [20], using the techniques described in [11, 12]. In particular, we created a single helper thread to run in parallel with EM3D (*i.e.*, the "compute thread"). The helper thread triggers cache misses normally incurred by the compute thread, but omits all computations unrelated to cache-miss generation.

The EM3D benchmark and its helper thread were executed on the MIT Graphite simulator [18]. Graphite was configured to simulate a single compute and partner core, with each core issuing 1 instruction per cycle in program order. The main core clocks at 1 GHz while the partner core clock is varied between 100 MHz–1 GHz to simulate different design points. Accessing the L1, L2, and main memory takes 3, 9, and 100 ns, respectively. Prefetch commands are communicated from the partner core to the main core via a FIFO queue; a prefetcher on the main core reads these commands and executes prefetches into the main core's L2 cache. However, there is no native hardware prefetcher in the main core. Finally, we assume a 1 GHz partner core consumes the same power as the main core, with partner core power reducing quadratically as its clock frequency decreases. In this paper we do not consider the additional power

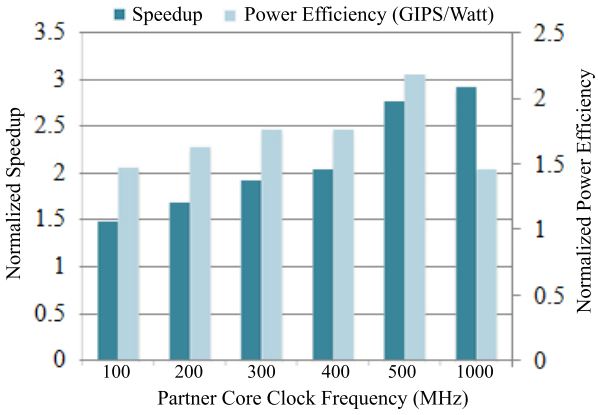

Figure 2: Speedup and power efficiency of EM3D with helper threads vs. execution without helper support.

savings that could be gained from simplifying the architecture of the partner core.

4.2 Experimental Results

Figure 2 presents our results. The dark bars report the application speedup with helper threads normalized against execution without helper threads. We see that a 1 GHz partner core nearly achieves a 3x performance gain, demonstrating that EM3D is memory bound, and that the helper thread can effectively tolerate cache-miss latency when running on a fast core. The figure also shows that helper threads can still provide a performance gain on slower partner cores. In fact, the performance gain for a 500 MHz partner core (2.7x) is still significant. Even with a 100 MHz partner core, the performance gain is still 1.5x. This demonstrates that the helper thread exhibits significant slack which can be used to tolerate slower Partner Core execution.

The light bars in Figure 2 report the power efficiency (performance/watt) of EM3D with helper threads normalized against power efficiency without helper threads. These bars show several interesting results. First, with a 1 GHz partner core, the normalized power efficiency is 1.46x. Even though the partner core doubles the system power in this case, power efficiency still improves because the helper thread provides a three-fold performance boost. Second, power efficiency is higher for the 100–500 MHz partner cores compared to the 1 GHz partner core. Although performance decreases on the slower partner cores, the performance loss is smaller than the corresponding power reduction. Finally, the 500 MHz partner core achieves the best normalized power efficiency, 2.2x. This is probably due to the slack exhibited by the helper thread, so very little performance loss occurs when slowing the partner core to 500 MHz. These results show that a partner core is capable of considerable performance gains even at low frequencies, where it can benefit from superior power efficiency.

5 Related Work

The Partner Core is most closely related to heterogeneous architectures that combine CPU and hardware accelerators on the same die. Historically, accelerators were designed as embedded solutions to specific problems such as graphics, encryption, or codec handling, but lately these systems have moved to include general-purpose accelerators like GPGPUs [17]. However, while an accelerator can share on-die resources, it still accrues the latency of communicating through a network. A key feature of the Partner Core concept is that it is tightly coupled to the main core, and so many of its resources are directly accessible. Sharing physical resources between cores directly is not a novel idea, but most existing architectures require significant routing wires and run into similar problems in latency and power [13].

Prefetching helper threads have been studied extensively in the past. Early work on helper threads investigated techniques for generating effective helper thread code for Simultaneous Multithreading (SMT) processors [2, 11, 24]. In addition to SMT processors, researchers have also investigated helper threads in the context of chip multiprocessors (CMPs) [15, 16]. Our work is similar to these techniques since we execute helper threads on a separate core from the compute threads. However, to the authors' knowledge, we are the first to use asymmetric cores to execute the helper and main computations.

6 Conclusion

The development of complex multicores has made manual optimization impossible for the average programmer. A potential solution is to employ algorithms that dynamically optimize performance, but many such systems utilize extra execution contexts, extra cores, or cycle stealing, incurring penalties that make it difficult to develop useful algorithms. In this paper, we introduce the concept of Partner Cores as a small, low-power core that is tightly coupled to the main application core in a parallel system. We showed that because of relaxed speed constraints, the technology is available for constructing partner cores that are 10% the size of each application core, and 10x lower in energy per instruction. We evaluated the Partner Core concept by running helper threads for memory prefetching, showing performance gains even at low frequencies.

References

[1] A.AGARWAL. Realizing a Power Efficient, Easy to Program Many Core: The Tile Processor. Presented at the Stanford Computer Systems EE380 Colloquium. Available online: http://www.stanford.edu/class/ee380/Abstracts/100203-slides.pdf.

[2] ANNAVARAM, M., PATEL, J. M., AND DAVIDSON, E. S. Data Prefetching by Dependence Graph Precomputation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture* (Goteborg, Sweden, June 2001), ACM.

[3] BARON, M. Low-Key Intel 80-Core Intro: The Tip of the Iceberg. *Microprocessor Report* (April 2007).

[4] BHAVNAGARWALA, A., KOSONOCKY, S., KOWALCZYK, S., JOSHI, R., CHAN, Y., SRINIVASAN, U., , AND WADHWA, J. A transregional CMOS SRAM with single, logic V and dynamic power rails. In *Symp. VLSI Circuits Dig. Tech. Papers* (2004), pp. 292–293.

[5] DALTON, M., KANNAN, H., AND KOZYRAKIS, C. Raksha: a flexible information flow architecture for software security. In *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture* (2007).

[6] DOWECK, J. Inside the CORE microarchitecture. In *presented at the 18th IEEE Hot Chips Symp.* (Palo Alto, CA, August 2006).

[7] GROCHOWSKI, E., AND ANNAVARAM, M. Energy per instruction trends in. Intel microprocessors. *Technology Intel Magazine 4*, 3 (2006), 1–8.

[8] HOFFMANN, H., EASTEP, J., SANTAMBROGIO, M. D., MILLER, J. E., AND AGARWAL, A. Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In *ICAC* (2010).

[9] HOFFMANN, H., SIDIROGLOU, S., CARBIN, M., MISAILOVIC, S., AGARWAL, A., AND RINARD, M. C. Dynamic knobs for responsive power-aware computing. In *ASPLOS* (2011), pp. 199–212.

[10] KIM, D., WEI LIAO, S. S., WANG, P., DEL CUVILLO, J., TIAN, X., ZOU, X., WANG, H., YEUNG, D., GIRKAR, M., AND SHEN, J. Physical Experimentation with Prefetching Helper Threads on Intel's Hyper-Threaded Processors. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization with Special Emphasis on Feedback-Directed and Runtime Optimization* (San Jose, CA, March 2004).

[11] KIM, D., AND YEUNG, D. Design and Evaluation of Compiler Algorithms for Pre-Execution. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, October 2002), ACM, pp. 159–170.

[12] KIM, D., AND YEUNG, D. A Study of Source-Level Compiler Algorithms for Automatic Construction of Pre-Execution Code. *ACM Transactions on Computer Systems 22*, 3 (August 2004).

[13] KUMAR, R., JOUPPI, N., AND TULLSEN, D. Conjoined-Core Chip Multiprocessing. In *Proc. Int'l Symp. Microarchitecture* (2004), IEEE CS Press, pp. 195–206.

[14] KWONG, J., RAMADASS, Y., VERMA, N., AND CHANDRAKASAN, A. A 65nm Sub-Vt Microcontroller with Integrated SRAM and Switched Capacitor DC-DC Convertor. *IEEE Journal of Solid-State Circuits 44*, 1 (January 2009), 115–125.

[15] LEE, J., JUNG, C., LIM, D., AND SOLIHIN, Y. Prefetching with Helper Threads for Loosely Coupled Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems 20*, 9 (July 2009).

[16] LU, J., DAS, A., HSU, W.-C., NGUYEN, K., AND ABRAHAM, S. G. Dynamic Helper Threaded Prefetching on the Sun UltraSPARC CMP Processor. In *Proceedings of the 38th International Symposium on Microarchitecture* (November 2005).

[17] LUEBKE, D., HARRIS, M., KRUGER, J., PURCELL, T., GOVINDARAJU, N., BUCK, I., WOOLLEY, C., AND LEFOHN, A. GPGPU: general purpose computation on graphics hardware. In *ACM SIGGRAPH 2004 Course Notes SIGGRAPH '04* (Los Angeles, CA, 2004), ACM.

[18] MILLER, J., KASTURE, H., KURIAN, G., III, C. G., BECKMANN, N., CELIO, C., EASTEP, J., AND AGARWAL, A. Graphite: A Distributed Parallel Simulator for Multicores. In *16th IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (January 2010).

[19] MUKHERJEE, S., KONTZ, M., AND REINHARDT, S. Detailed Design and Evaluation of Redundant Multithreading Alternatives. In *ISCA '02: Proceedings of the 29th Annual International Symposium on Computer Architecture* (2002).

[20] ROGERS, A., CARLISLE, M., REPPY, J., AND HENDREN, L. Supporting Dynamic Data Structures on Distributed Memory Machines. *ACM Transactions on Programming Languages and Systems 17*, 2 (March 1995).

[21] TAYLOR, M. B., LEE, W., MILLER, J. E., WENTZLAFF, D., BRATT, I., GREENWALD, B., HOFFMANN, H., JOHNSON, P., KIM, J., PSOTA, J., SARAF, A., SHNIDMAN, N., STRUMPEN, V., FRANK, M., AMARASINGHE, S., AND AGARWAL, A. Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *ISCA '04: Proc of the 31st annual International Symposium on Computer Architecture* (June 2004), pp. 2–13.

[22] WENTZLAFF, D., GRIFFIN, P., HOFFMANN, H., BAO, L., EDWARDS, B., RAMEY, C., MATTINA, M., MIAO, C.-C., BROWN, J. F., AND AGARWAL, A. On-chip interconnection architecture of the Tile processor. *IEEE Micro 27*, 5 (Sept-Oct 2007), 15–31.

[23] ZHAI, B., NAZHANDALI, L., OLSON, J., REEVES, A., MINUTH, M., HELFAND, R., PANT, S., BLAAUW, D., , AND AUSTIN, T. A 2.60 pJ/Inst subthreshold sensor processor for optimal energy efficiency. In *Symp. VLSI Circuits Dig. Tech. Papers* (June 2006), pp. 154–155.

[24] ZILLES, C., AND SOHI, G. Execution-Based Prediction Using Speculative Slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture* (Goteborg, Sweden, June 2001).