A Mechanism for Energy-efficient Reuse of Decoding and Scheduling of x86 Instruction Streams

Marcelo Brandalero, Antonio Carlos S. Beck
Instituto de Informtica
Universidade Federal do Rio Grande do Sul
Porto Alegre, Brazil
{mbrandalero, caco}@inf.ufrgs.br

***Abstract*—Current superscalar x86 processors decompose each CISC instruction (variable-length and with multiple addressing modes) into multiple RISC-like µops at runtime so they can be pipelined and scheduled for concurrent execution. This challenging and power-hungry process, however, is usually repeated several times on the same instruction sequence, inefficiently producing the very same decoded and scheduled µops. Therefore, we propose a transparent mechanism to save the decoding and scheduling transformation for later reuse, so that next time the same instruction sequence is found it can automatically bypass the costly pipeline stages involved. We use a coarse-grained reconfigurable array as a means to save this transformation, since its structure enables the recovery of µops already allocated in time and space, and also larger ILP exploitation than superscalar processors. The technique can reduce the energy consumption of a powerful 8-issue superscalar by 31.4% at low area costs, while also improving performance by 32.6%.**

***Index Terms*—x86; superscalar; dynamic optimization; instruction-level parallelism.**

I. Introduction

The domain of x86 processors over the general-purpose computing market has demonstrated the importance of maintaining binary compatibility to allow the execution of software already deployed in new processors. However, implementing a complex (CISC) ISA such as x86 is challenging, because there are over a thousand instructions with variable lengths and addressing modes.

To cope with this, x86 processors have long been designed with a decoder that decomposes each x86 instruction into simpler RISC-like operations named µops [1] [2]. This step allows executing x86 in a pipelined organization, enabling dynamic scheduling and superscalar execution of µops to exploit high amounts of instruction-level parallelism (ILP). However, this whole process is very expensive: the decoding of each x86 instruction to multiple µops and their dynamically scheduling for concurrent execution require complex structures and logic. Previous works report that decode can account for up to 10% of the total package power [3], and scheduling for 10-20% [4] [5]. Our own experiments, using a methodology that will be described later, have found similar results (Fig. 1).

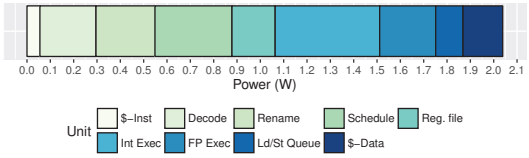

Fig. 1. Power breakdown of a modern superscalar processor. The structures used for decode and schedulling account for 27.5% of the core's power.

To reduce the energy costs, recent Intel processors already store decoded µops in a special L0 cache inside the pipeline [6] [7], so that repeating instruction sequences need not be decoded multiple times. As can be seen in Fig. 2 (a and b), this mechanism has been moving deeper into the pipeline, improving the amount of processing that is saved. While previous works have proposed efficient implementations of the scheduling logic [4] [8], no work has yet proposed to move one step further (Fig. 2c) and store the already scheduled µops inside the pipeline. By doing so, repeating instruction sequences can be automatically decoded and scheduled in a single step, skipping the complex pipeline stages that are involved, and improving energy consumption and performance.

The contributions of this paper are twofold:

- We propose a new x86 processor design that reduces the utilization of the complex hardware structures responsible for decoding and scheduling of repeating instruction sequences, by saving this processing in a special cache and reusing it afterward, thereby improving energy con-

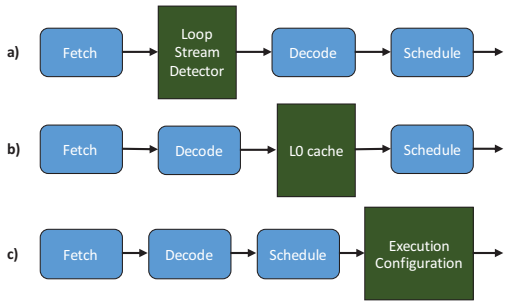

Fig. 2. **(a)** Loop Stream Detector, in the Penryn µarch (2007). **(b)** L0 cache, in the Sandy Bridge µarch (2011). **(c)** Execution configuration, in this work.

978-3-9815370-8-6/17/$31.00 ©2017 IEEE

sumption.
- We use a transparent coarse-grained reconfigurable array (CGRA) as a means to implement this reuse, taking advantage of its intrinsic structure to enable the recovery of μops already allocated in time and space. Since it is also capable of efficiently exploiting ILP, performance improves.

To evaluate the proposed system, we use gem5 [9], McPAT [10], CACTI [11] and Cadence RTL compiler to compare the performance, area and energy against an 8-issue superscalar x86 processor based on the Core i7 Haswell microarchitecture (one of the latest from Intel [12]). The results show that we can speed up such a powerful processor by 32.6% and reduce its energy consumption by 31.4% in a considerable number of benchmarks while introducing less than 12.5% area overhead.

The remainder of this article is organized as follows. Section 2 presents related work. Section 3 describes the proposed mechanism. Section 4 presents performance, area, power and energy results. Section 5 discusses and concludes this work.

II. Related Work

The complexity of superscalar processors has long concerned the designers. In these systems, dynamic scheduling is used with out-of-order execution to exploit high amounts of ILP by selecting independent instructions for execution. However, the number of transistors used to implement part of this structure, named instruction window, has a direct impact on the energy consumption and grows quadratically with its size and the number of operations selected concurrently [8] [13].

There are two approaches to improving the efficiency of these processors. The first one modifies the baseline microarchitecture, in particular, the instruction window, by reducing the number of comparisons required to select instructions. This can be done by grouping instructions by dependence [8], which also enables fusing them for single-cycle execution [14], or by modifying the size of the instruction window dynamically to eliminate redundant lookups [4].

The second approach involves designing a new, more efficient microarchitecture that implements an optimized ISA. However, this design freedom breaks binary compatibility and therefore requires some code transformation technique, such as binary translation [15], to allow the execution of native programs in the new system, as discussed next.

The Transmeta Crusoe system uses a VLIW processor to execute x86 instructions [16]. A binary translation system named Code Morphing Software (CMS) inspects x86 code at runtime and transforms it to VLIW instructions, saving this transformation in a cache.

The Warp Processor [17] uses a simplified FPGA that is coupled to an ARM7 and a small CAD processor. While the program executes in the main processor, the CAD processor analyses and transforms entire sequences of instructions into FPGA configurations. These configurations are used to set up the FPGA and execute the instructions next time.

The Configurable Compute Accelerator (CCA) [18] is a configurable matrix of functional units that is also coupled to an ARM processor. At runtime, application kernels are discovered using a graph analyzer and transformed into instructions for the CCA, which replace the CPU instructions in the system's trace cache.

The Dynamic Instruction Merging (DIM) system [19] is a CGRA that is tightly coupled to a MIPS processor. Using a simplified hardware binary translation algorithm, application kernels are dynamically transformed into CGRA configurations and saved in a special cache. Unlike the CCA, the system accesses the special cache using the program counter (PC) of the application kernels to check if they can execute in the array.

DynaSpAM [20] is a reconfigurable accelerator coupled to a non-x86 out-of-order superscalar pipeline. The system modifies the scheduler logic to simultaneously allocate instructions for execution in the pipeline units and transform the code sequence for execution in the reconfigurable fabric.

Our approach:

- Unlike all previous systems (except CMS), it is the only one that accelerates x86 (the ISA used in nearly all general-purpose computers) running in a superscalar core;
- Unlike CMS, our code transformation algorithm is implemented in hardware and processes μops; therefore, the overheads of the technique are much smaller;
- Unlike the Warp Processor, which uses an FPGA with high configuration latency, our system can accelerate any sort of application by using a CGRA with small configuration times;
- Unlike CCA and DIM, our CGRA is tightly coupled to a superscalar core and can exploit more ILP than DynaSpAM.

In summary, the technique proposed is the only one that uses a CGRA tightly-coupled to an x86 superscalar and, therefore, is able to skip the complex pipeline stages of decoding and scheduling, improving any application's performance and energy consumption.

III. Proposed System

An overview of the proposed system is presented in Fig. 3. There are five blocks: the baseline processor, the code transformation (CT) module, the configuration cache, the CGRA and the results buffer. The process works as follows: each instruction sequence executes for the first time in the x86 pipeline (regular execution, in the figure). As it executes, it is transformed by the CT module into CGRA configurations and saved in the configuration cache. Next time the same sequence is found again in the fetch stage, the configuration is loaded from the configuration cache and the sequence is executed more efficiently in the CGRA (optimized execution), bypassing the expensive process of x86 decode and μop scheduling.

We describe first the hardware of the system and then the process to transform x86 code to execute in the CGRA.

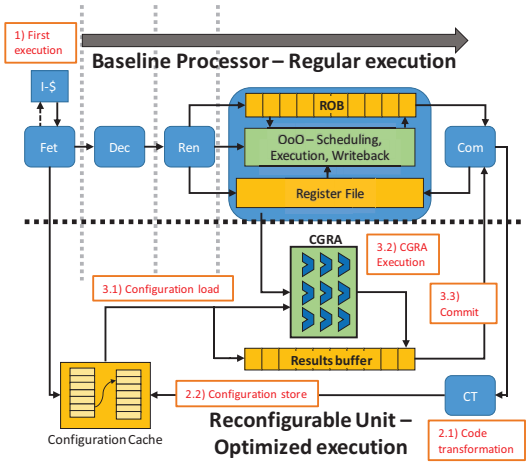

Fig. 3. Overview of the proposed system.

A. Hardware

The baseline processor is an 8-issue superscalar x86 processor with the parameters given in Table I. This organization closely resembles the modern Intel Haswell microarchitecture, one of the latest Intel Core i7 processors, which is implemented in 22nm [12].

The reconfigurable array, detailed in Fig. 4, is a heterogeneous matrix of functional units (FUs) composed entirely of combinational logic and divided into rows and columns. Data propagates from left to right, so each FU occupies a row and a sequence of columns depending on its latency. The simplest units are ALUs, which take $\frac{1}{3}$ of a processor cycle in our implementation and correspond to a single column. Therefore, we define a level, the equivalent of one processor cycle, as a sequence of 3 columns. The latencies and FU distribution can be customized depending on the process technology and design constraints; we describe our design in Table II.

The communication between FUs is done using context lines, which are initially fed by values from the register file through the input context. Before each column, a crossbar network (X in the figure) selects the context line that feeds each functional unit. After the operation, the crossbar selects the value that will be propagated to the next column through the context lines and also routes the output to the results buffer, where up to 192 results can be temporarily stored.

The Code Transformation (CT) module is a 4-stage pipelined unit that transforms sequences of µops into CGRA configurations, as will be explained in detail in the next section. The configuration cache can hold up to 256 configurations and consists of two tables: the first one stores the PCs of translated sequences and is used to quickly check if a configuration exists (will be discussed in detail next): the second table holds the configurations, and is accessed only when one must be loaded and executed. Both these designs were adapted and extended from [21] for better integration with the superscalar core.

TABLE I
BASELINE PROCESSOR PARAMETERS.

Pipeline: 8-wide out-of-order, with 4 ALU ports, 2 mult. ports, 2 load ports and 1 store port. Instr. queue: 60 µops. Load buffer: 72 µops. Store buffer: 42 µops. ROB entries: 192 µops. Memory dependence prediction via store sets.
L1 D+I caches: 32kB each. 8-way set associative, 2 cycles hit latency.
L2 cache: 256kB, 8-way associative, 8 cycles hit latency.
L3 cache: 2MB, 16-way associative, 18 cycles hit latency.

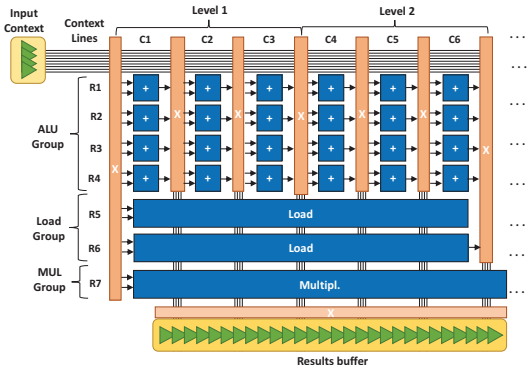

Fig. 4. The CGRA in detail.

B. Code transformation and execution

The process described next follows the steps illustrated in Fig. 3.

1) First execution in the baseline processor: x86 instructions are fetched from the instruction cache and decoded into one or more µops. In the rename stage, the logical registers encoded in the µops are renamed to physical registers, eliminating false data dependencies. Next, µops are copied following program order to the reorder buffer (ROB), a structure used to hold temporary operation results. This is the pipeline front-end, where operations are processed in-order.

These µops are dynamically scheduled for out-of-order execution in the back-end as their input registers become ready. The outputs are written to the ROB and forwarded to the µops waiting in the scheduling queue for the result. In the commit stage, the results are processed from the ROB and written back to the register file following program order to preserve true data dependencies.

Memory operations are handled by dedicated load and store queues which avoid pipeline stalls and enable memory optimizations, such as load bypassing and load forwarding [22]. The latter technique allows a load operation that accesses the same address as a previous store waiting in the queue to read its value directly, bypassing a cache access.

2) Code transformation: In parallel with the execution previously described, the CT module processes sequences of

TABLE II
CGRA PARAMETERS.

ALUs	12 per level, with a latency of $\frac{1}{3}$ of a cycle and organized as three columnns with four ALU rows each.
Multipliers	2 rows per level, with a latency of 3 cycles.
Load Units	2 rows per level, with a latency of 2 cycles.
Store Units	1 row per level, with a latency of 1 cycles.

µops by using a greedy algorithm to allocate them to the functional units in the CGRA.

The process starts when the first application basic block (BB) commits. Each new µop is allocated in the lowest level where its inputs are ready, and a functional unit is available. It stops when either one occurs:

- an unsupported µop is detected (such as a floating point operation);
- the configuration is full (all available levels are occupied);
- a maximum number of µops or BBs have been mapped;

When this process completes, a configuration for the CGRA is generated and stored in the configuration cache, along with the PC of the first µop transformed. Each configuration contains the operations of the functional units, the crossbar setup and the logical destination registers of each µop in the sequence so that the register file may be updated later on.

To maximize ILP, the algorithm exploits two forms of speculation using simple prediction algorithms.

- **Control speculation**: the algorithm speculates on the sequence of BBs that will be executed by mapping multiple BBs to the same configuration, allowing ILP exploitation across control boundaries. The number of BBs per configuration is the *trace length*, which defines how aggressive the speculation is.
- **Memory dependence speculation**: the algorithm speculates on the dependence between memory operations by executing load operations earlier than preceding stores. The prediction is that if a memory address collision has not occurred in the translated sequence then it will not occur again.

Misspeculations during execution will cause invalid operations to be flushed and the configuration to be flagged for removal, as will be explained next.

3) Configuration load and execution: When the baseline processor detects a new BB during regular execution, the fetch unit looks it up in in the configuration cache. If the block is not found, then it has not been transformed, and therefore must be fetched from the instruction cache and regularly executed in the baseline processor. Otherwise, the associated CGRA configuration is loaded: the FUs and crossbars are set up and the results buffer is prepared with the logical registers that will be written by each µop. A synchronization µop is inserted in the ROB to mark the code sequence that will execute in the CGRA. This entire load process replaces the costly decoding of x86 instructions and scheduling of µops, enabling large energy savings. It may take up to 8 cycles (the number of front-end pipeline stages in the baseline processor) with no performance impact because, meanwhile, the baseline processor will still be executing previous µops.

Execution in the CGRA starts when the configuration is completely loaded. Data propagates from the input and is processed by the functional units. The crossbars redirect the output of each functional unit to the corresponding results buffer entry, where the results are temporarily stored. When the synchronization µop previously inserted in the main ROB reaches its head, the commit stage in the baseline processor starts committing µops from the results buffer. Each µop is then allocated to a physical register where its result will be stored and finally committed.

The commit stage detects invalid speculations by checking the memory addresses accessed by each load/store and also the results of control µops. In case a misspeculation is found, the last valid µop is marked in the results buffer, so that the following ones will not be committed, and a misspeculation flag is set for the configuration in the cache. If the same happens again when the configuration executes next time, the configuration is removed from the cache; otherwise, the flag is cleared.

IV. Results

A. Methodology

To evaluate the performance, we modified the out-of-order processor in gem5 [9] to implement the code transformation algorithm, the configuration cache, and the CGRA. We used the parameters shown in Table I to model the baseline processor, which is based a recent Intel Haswell [12]. We used McPAT [10] for area and energy; Cadence RTL Compiler to synthesize the CGRA; and estimated the configuration cache size using CACTI [11] - all considering a 22 nm process technology. We experimented with multiple CGRA designs by varying the number of levels (15, 30 and 60) and the trace length (from one to 10 BBs per configuration - larger ones were tested but provided high rates of mispredictions).

The proposed system was evaluated using a subset of 9 benchmarks from the Mibench suite [23], all compiled using gcc 5.3.0 with the -O3 optimization flag. We present here a characterization of these benchmarks, showing that they cover a broad range of applications with distinct dynamic behaviors.

In Fig. 5, we ordered each application's BBs increasingly by their contribution to the execution time (coverage) and show how many BBs are required to achieve a particular coverage rate and also their average size. Some applications, such as *susans*, have an avg. BB size of 22 µops and a single very distinct kernel that covers 89% of the application, requiring four more BBs to achieve 98% coverage. Other applications, such as *bitcount*, have many distinct kernels (with a smaller avg. size of 8.4 µops), with the most significant one covering 33% and requiring 16 additional ones to cover 98%.

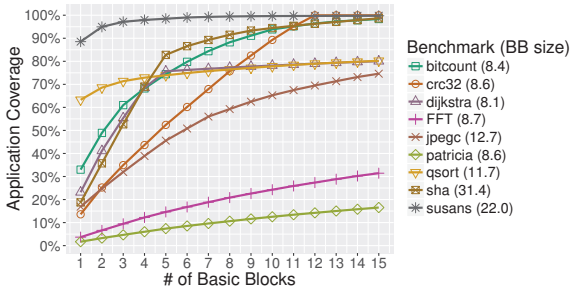

Fig. 5. Dynamic behavior and avg. BB size of each benchmark.

Applications with smaller BBs are typically more difficult to accelerate, because they need better control prediction mechanism, as is the case with applications with too many kernels. We will show that our approach, just like the superscalar processor, can accelerate any application.

B. Results

1) Performance: We present in Fig. 6 the geomean application speedup against the superscalar processor for each combination of trace length and number of levels. As can be seen, the speedup increases with both parameters, because more speculation allows better exploiting ILP across BBs, and more levels allow it to support larger instruction sequences that amortize the reconfiguration costs. A highest mean speedup of 32.6% is achieved in the best case, with a 60-level design. The only slowdown occurs when not using speculation because, in this case, only the baseline processor can execute µops from multiple BBs simultaneously. The results show that the design with 30 levels achieves the best trade-off, given that it enables enough µops to be allocated considering the degree of speculation exploited.

Considering benchmark-specific execution in the 30-level design, Fig. 7 presents the speedup and Table III additional results. There are significant improvements in nearly all applications, because of the CGRA's ability to speed up chains of data-dependent operations, thereby exploiting more ILP than the superscalar. *Bitcount* is accelerated the most by 120.9%, because of the high rate of ALU operations (83.7%) and the low rate of PC (µops per cycle) in the baseline processor (2.4), which indicates many dependencies among these operations.

The only exceptions are *crc32* and *susans*. In these applications, there are many memory dependencies, and the critical path in a configuration lies in chains of load operations. However, the baseline processor can accelerate these chains by using the load/store queue; the code transformation module automatically detects that this is the case and does not save the configuration. Therefore, the coverage (rate of µops that execute in the CGRA) for these applications is small, and there are no performance losses. Additionally, *crc32* achieves the highest PC in the baseline processor (5.1) and the avg. configurations in *susans* have the longest execution time (27 cycles) of all appplications.

2) Energy: Fig. 1 presents a power breakdown of the superscalar processor. We amortize the costs of decode, scheduling,

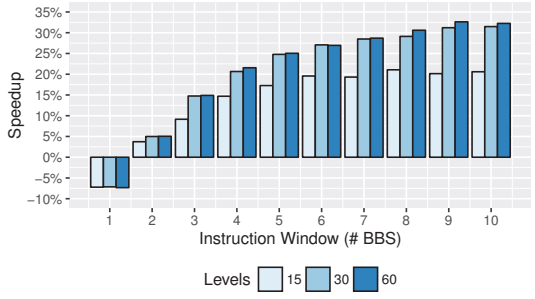

Fig. 6. Geomean sytem speedup for distinct number of levels and trace lengths. The text labels show the speedup with 60 levels.

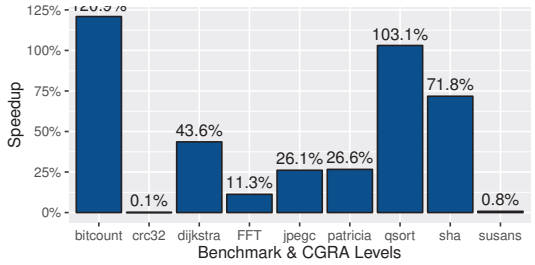

Fig. 7. Speedup achieved by each benchmark with the 30-level design.

as was previously stated, and also the ones associated with the load/store queue and integer execution because memory operations are allocated only once when the configuration is generated and execution is more efficient in the CGRA.

Fig. 8 shows the energy consumed by each benchmark in the baseline processor and in the proposed system, separating the consumption of the superscalar core, the CGRA and the configuration cache. The results are positive in almost all cases, and a geomean reduction of 31.4% from the baseline is achieved.

Two factors enable these gains. First, as already explained, we bypass the complex pipeline stages involved with instruction decoding and µop scheduling, therefore reducing the energy that the superscalar consumes (*Superscalar* bars in the figure). Instead, this information is fetched from the configuration cache (*CfgCache* bars), and the code sequence is executed in the CGRA (*CGRA* bars). Second, we reduce the application execution time and, therefore, applications with large speedups (such as *bitcount* and *qsort*) also achieve the most significant energy reductions. There are only two cases with marginal increases, which are *crc32* and *susans*. As was previously stated, coverage for these benchmarks is small, so there is no speedup. However, looking up configurations in the configuration cache add a low energy overhead to these benchmarks.

3) Area: The CGRA and the configuration cache have areas of 4.18mm^2 and 1.34mm^2, respectively. The superscalar core occupies 13.94 mm^2. Because of this small size, modern processor contain multiple cores and complex graphics processing units (GPUs) in the same die. Compared to Haswell 4770K, which occupies 177mm^2 and has four cores [24], our design introduces only 12.5% area overhead when implementing the technique in each of the cores.

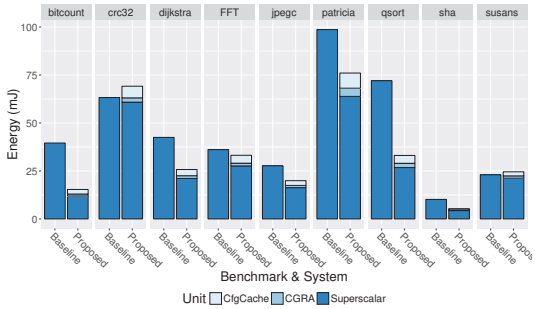

Fig. 8. Energy consumption by each benchmark with the 30-level design.

V. Conclusions and future work

This paper presented a means to save the decoding and scheduling of x86 instructions for reuse by coupling a CGRA to the superscalar pipeline. The technique can reduce the energy consumption of a modern processor by 31.4% and improve its performance by 32.6% while adding only 12.5% area.

As future work, we are currently working on expanding to multicore processors and improving the code transformation algorithm for better selecting the code regions to transform.

References

[1] Intel, "Intel Architecture Optimization Manual," 1997. [Online]. Available: http://www.intel.com/design/pentium/MANUALS/24281603.pdf

[2] I. C. Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Desktop Platforms Group, G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, P. Roussel, and I. C. Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Desktop Platforms Group, "The Microarchitecture of the Pentium 4 Processor," *Intel Technology Journal*, vol. 1, no. 1, pp. 1–13, 2001.

[3] M. Hirki, Z. Ou, K. N. Khan, J. K. Nurminen, and T. Niemi, "Empirical Study of the Power Consumption of the x86-64 Instruction Decoder," in *USENIX Workshop on Cool Topics on Sustainable Data Centers (CoolDC 16)*, 2016.

[4] D. Folegnani and A. Gonzalez, "Energy-effective issue logic," in *Proceedings 28th Annual International Symposium on Computer Architecture*. IEEE Comput. Soc, 2001, pp. 230–239.

[5] C. Isci and M. Martonosi, "Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data," in *Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36'03)*. IEEE Computer Society, dec 2003, p. 93.

[6] M. Dixon, P. Hammarlund, S. Jourdan, and R. Singhal, "The Next Generation Intel® Core Microarchitecture," *Intel Technology Journal*, vol. 14, no. 3, pp. 8–28, 2010.

[7] L. Gwennap, "Sandy Bridge Spans Generations," *Microprocessor Report*, no. 328, p. 8, 2010.

[8] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," in *Proceedings of the 24th annual international symposium on Computer architecture*. ACM, 1997, pp. 206–218.

[9] N. Binkert, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, D. A. Wood, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, and T. Krishna, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, p. 1, aug 2011.

[10] Sheng Li, J.-H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-42.*, 2009, pp. 469–480.

[11] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A Tool to Model Large Caches," HP Laboratories, Tech. Rep., 2009.

[12] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton, "Haswell: The Fourth-Generation Intel Core Processor," *IEEE Micro*, vol. 34, no. 2, pp. 6–20, mar 2014.

[13] J. L. Henessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann, 2011.

[14] G. Alolu, Z. Jin, M. Köksal, O. Javeri, and S. Önder, "LaZy superscalar," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture - ISCA '15*, 2015, pp. 260–271.

[15] E. Altman, D. Kaeli, and Y. Sheffer, "Welcome to the opportunities of binary translation," *Computer*, vol. 33, no. 3, pp. 40–45, mar 2000.

[16] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The Transmeta Code Morphing Software: using speculation, recovery, and adaptive retranslation to address real-life challenges," pp. 15–24, mar 2003.

[17] R. Lysecky, G. Stitt, and F. Vahid, "Warp Processors," *ACM Transactions on Design Automation of Electronic Systems*, vol. 11, no. 3, pp. 659–681, jul 2006.

[18] N. Clark, M. Kudlur, S. Mahlke, and K. Flautner, "Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization," in *37th International Symposium on Microarchitecture (MICRO-37'04)*. IEEE, 2004, pp. 30–40.

[19] A. C. S. Beck, M. B. Rutzig, G. Gaydadjiev, and L. Carro, "Transparent Reconfigurable Acceleration for Heterogeneous Embedded Applications," in *Proceedings of the conference on Design, automation and test in Europe - DATE '08*. New York, New York, USA: ACM Press, mar 2008, pp. 1208–1213.

[20] F. Liu, H. Ahn, S. R. Beard, T. Oh, and D. I. August, "DynaSpAM : Dynamic Spatial Architecture Mapping using Out of Order Instruction Schedules," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*, 2015, pp. 541–553.

[21] A. C. S. Beck, M. B. Rutzig, and L. Carro, "A transparent and adaptive reconfigurable system," *Microprocessors and Microsystems*, vol. 38, no. 5, pp. 509–524, jul 2014.

[22] J. P. Shen and M. H. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*, 1st ed. Waveland Press, 2013.

[23] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. IEEE, 2001, pp. 3–14.

[24] A. L. Shimpi, "The Haswell Review: Intel Core i7-4770K & i5-4670K Tested," 2013. [Online]. Available: http://www.anandtech.com/show/7003/the-haswell-review-intel-core-i74770k-i54560k-tested/5

TABLE III
Execution results in the design with 30 levels.

Benchmark	Optype Breakdown				Baseline µPC	Avg. Configuration			Speedup	Coverage
	ALU	Load	Store	Branches		µops	BBs	Cycles		
FFT	63.1%	12.4%	7.5%	11.4%	3.3	61.7	7.6	12.8	1.11	44.0%
bitcount	83.7%	3.4%	0.8%	11.8%	2.4	66.1	7.8	11.2	2.21	94.9%
crc32	61.2%	18.4%	8.7%	11.7%	5.1	74.9	9.0	14.0	1.00	11.3%
dijkstra	71.4%	10.6%	5.7%	12.3%	3.3	69.9	8.7	13.8	1.44	81.4%
jpegc	65.1%	17.5%	8.3%	7.9%	2.2	60.5	4.6	17.7	1.26	65.4%
patricia	66.7%	11.7%	7.6%	11.6%	2.2	53.6	6.4	11.7	1.27	51.4%
qsort	65.0%	13.3%	11.2%	8.6%	1.7	93.6	7.4	18.2	2.03	69.5%
sha	75.0%	11.4%	4.3%	3.2%	2.3	71.0	2.5	11.7	1.72	70.3%
susans	73.8%	12.7%	0.1%	4.5%	2.6	69.9	3.0	27.0	1.01	19.3%