Dynamic Vectorization in the E2 Dynamic Multicore System

Dynamic Vectorization in the E2 Dynamic Multicore Architecture

To appear in the proceedings of HEART 2010

Andrew Putnam
Microsoft Research
anputnam@microsoft.com

Aaron Smith
Microsoft Research
aasmith@microsoft.com

Doug Burger
Microsoft Research
dburger@microsoft.com

ABSTRACT

Previous research has shown that Explicit Data Graph Execution (EDGE) instruction set architectures (ISA) allow for power efficient performance scaling. In this paper we describe the preliminary design of a new dynamic multicore processor called E2 that utilizes an EDGE ISA to allow for the dynamic composition of physical cores into logical processors. We provide details of E2's support for dynamic reconfigurability and show how the EDGE ISA facilities out-of-order vector execution.

Categories and Subject Descriptors

C.1.2 [**Computer Systems Organization**]: Multiple Data Stream Architectures—*single-instruction-stream, multiple-data-stream processors (SIMD), array and vector processors*; C.1.3 [**Computer Systems Organization**]: Other Architecture Styles—*adaptable architectures, data-flow architectures*

General Terms

Design, Performance

Keywords

Explicit Data Graph Execution (EDGE)

1. INTRODUCTION

Chip designers have long relied on dynamic voltage and frequency scaling (DVFS) to trade off power for performance. However, voltage scaling no longer works as processors approach the minimum threshold voltage (V_{min}) as frequency scaling at V_{min} reduces both power and performance linearly, achieving no reduction in energy. Power and performance trade-offs are thus left to either the microarchitecture or the system software.

When designing an architecture with little (if any) DVFS, designers must choose how to spend the silicon resources. Hill and Marty [6] described four ways that designers could use these resources: (1) many small, low performance, power efficient cores, (2) few large, power inefficient, high performance cores, (3) a heterogeneous mix of both small and large cores, and (4) a dynamic architecture capable of combining or splitting cores to adapt to a given workload. Of these alternatives, the highest performance and most energy efficient design is the dynamic architecture. Hill and Marty characterized what such a dynamic processor could do but did not describe the details of such an architecture.

TFlex [9] is one proposed architecture that demonstrated a large dynamic range of power and performance by combining power efficient, lightweight processor cores into larger, more powerful cores through the use of an Explicit Data Graph Execution (EDGE) instruction set architecture (ISA). TFlex is dynamically configurable to provide the same performance and energy efficiency as a small embedded processor or to provide the higher performance of an out-of-order superscalar on single-threaded applications.

Motivated by these promising results, we are currently designing a new dynamic architecture called E2 that utilizes an EDGE ISA to achieve high performance power efficiently [3]. The EDGE model divides a program into blocks of instructions that execute atomically. Blocks consist of a sequence of dataflow instructions that explicitly encode relationships between producer-consumer instructions, rather than communicating through registers as done in a conventional ISA. These explicit encodings are used to route operands to private reservation stations (called *operand buffers*) for each instruction. Registers and memory are only used for handling less-frequent inter-block communication.

Prior dynamic architectures [7, 9] have demonstrated the ability to take advantage of task and thread-level parallelism, but handling data-level parallelism requires dividing data into independent sets and using thread-level parallelism. In this paper we focus on efficiently exploiting data-level parallelism, even without threading, and present our preliminary vector unit design for E2. Unlike previous in-order vector machines, E2 allows for out-of-order execution of both vectors and scalars.

The E2 instruction set and execution model supports three new capabilities that enable efficient vectorization across a broad range of codes. First, by slicing up the statically programmed issue window into vector lanes, highly concurrent, out-of-order issue of mixed scalar and vector operations can be achieved with lower energy overhead than scalar mode. Second, the statically allocated reservation stations permit the issue window to be treated as a vector register file, with wide fetches to memory and limited copying between a vector load and the vector operations. Third, the atomic block-based model in E2 permits refreshing of vector (and scalar) instruction blocks mapped to reservation stations, enabling repeated vector operations to issue with no fetch or decode energy overhead after the first loop iteration. Taken together, these optimizations will reduce the energy associated with finding and executing many sizes of vectors across a wide range of codes.

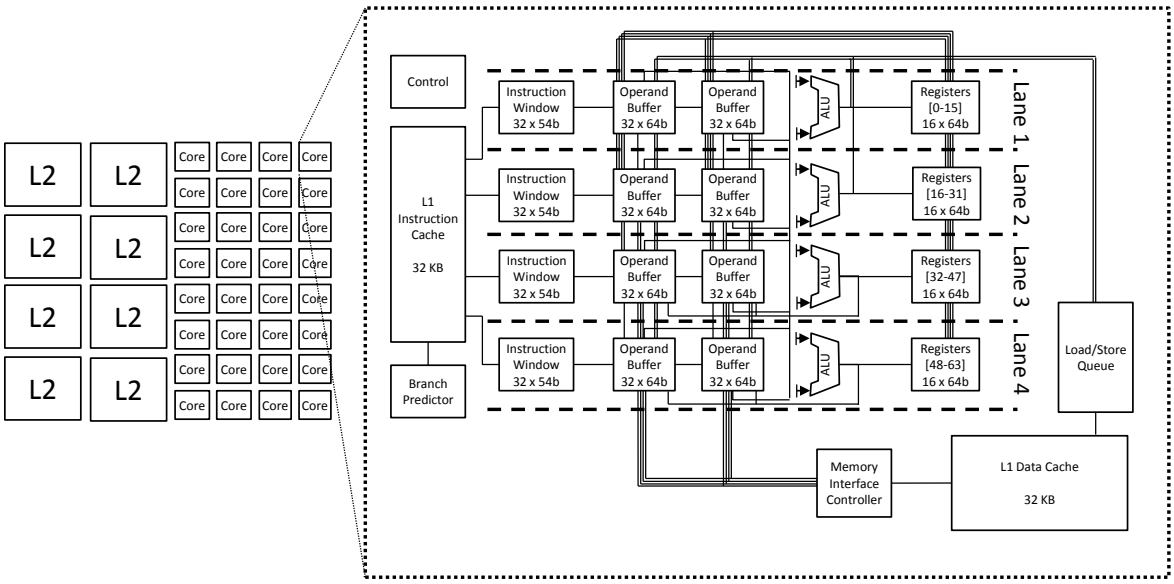

Figure 1: E2 microarchitecture block diagram. In vector mode, each core is composed of four independent vector lanes, each with a 32-instruction window, two 64-bit operand buffers, an ALU for both integer and floating point operations, and 16 registers. In scalar mode, the ALUs in lanes 3 and 4 are powered down, and the instruction windows, operand buffers, and registers are made available to the other two lanes.

2. THE E2 ARCHITECTURE

E2 is a tiled architecture that consists of low power, high performance, decentralized processing cores connected by an on-chip network. This design provides E2 with the benefits of other tiled architectures - namely simplicity, scalability, and fault tolerance. Figure 1 shows the basic architecture of an E2 processor containing 32 cores, and a block diagram of the internal structure of one physical core.

A core contains N lanes (in this paper we choose four), with each lane consisting of a 64-bit ALU and one bank of the instruction window, operand buffers, and register file. ALUs support both integer and floating point operations, as well as fine-grained SIMD execution (eight 8-bit, four 16-bit, or two 32-bit integer operations per cycle, or two single-precision floating point calculations per cycle). This innovation of breaking the window into lanes allows for high vector throughput with little additional hardware complexity.

E2's EDGE ISA restricts blocks in several ways to simplify the hardware that maps blocks to the execution substrate and detect when blocks are finished executing. Blocks are variable-size: they contain between 4 and 128 instructions and may execute at most 32 loads and stores. The hardware relies on the compiler to break programs into blocks of dataflow instructions and assign load and store identifiers to enforce sequential memory semantics [12]. To improve performance, the compiler uses predication to form large blocks filled with useful instructions. To simplify commit, the architecture relies on the compiler to ensure that a single branch is produced from every block, and to encode the register writes and the set of store identifiers used.

E2 cores operate in two execution modes: *scalar* mode and *vector* mode. In scalar mode, any instruction can send operands to any other instruction in the block, and all but two of the ALUs are turned off to conserve power. In vector mode, all *N* ALUs are turned on, but instructions can only send operands to instructions in the same vector lane. The mode is determined on a per-block basis from a bit in the block header. This allows each core to adapt quickly to different application phases on a block-by-block basis.

2.1 Composing Cores

One key characteristic that distinguishes E2 from other processors is the ability to dynamically adapt the architecture for a given workload by composing and decomposing cores. Rather than fixing the size and number of cores at design time, one or more *physical* cores can be merged together at runtime to form larger, more powerful *logical* cores. For example, serial portions of a workload can be handled by composing every physical core into one large logical processor that performs like an aggressive superscalar. Or, when ample thread-level parallelism is available, the same large logical processor can be split so each physical processor can work independently and execute instruction blocks from independent threads. Merging cores together is called *composing* cores, while splitting cores is called *decomposing* cores.

Logical cores interleave accesses to registers and memory among the physical cores to give the logical core the combined computational resources of all the composed physical cores. For example, a logical core composed of two physical cores uses an additional bit of the address to choose between the two physical caches, effectively doubling the L1 cache capacity. The register files are similarly interleaved, but since only 64 registers are supported by the ISA, the additional register file capacity is powered gated to reduce power consumption.

Each instruction block is mapped to a single physical pro-

cessor. When composed, the architecture uses additional cores to execute speculative instruction blocks. When the non-speculative block commits, it sends the commit signal along with the exit branch address to all other cores in the logical processor. Speculative blocks on the correct path continue to execute, while blocks on non-taken paths are squashed. Details of this process are discussed further in section 2.2.1.

Core composition is done only when the overhead of changing configurations is outweighed by the performance gains of a more efficient configuration. Composition is always done at block boundaries and is initiated by the runtime system. To increase the number of scenarios in which composition is beneficial, E2 provides two different ways to compose cores, each offering a different trade-off in overhead and efficiency.

Full composition changes the number of physical cores in a logical core, and changes the register file and cache mappings. Dirty cache lines are written out to main memory lazily. Logical registers and cache locations are distributed evenly throughout the physical cores. Cache lines are mapped via a simple hash function, leading to a larger logical cache that is the sum of the cache capacities of all physical cores.

Quick composition adds additional cores to a logical processor, but retains the same L1 data cache and register mappings, and does not write dirty cache lines out to main memory. This leaves the logical processor with a smaller data cache than possible with full composition, but ensures that accesses to data already in the cache will still hit after composing. Quick composition is the most useful for short-lived bursts of activity where additional execution units are useful, but where the overhead of reconfiguring the caches is greater than the savings from a larger, more efficient cache configuration.

Decomposition removes physical cores from a logical processor and powers the removed cores down to conserve energy. Execution continues on the remaining physical cores. Decomposition requires flushing the dirty lines of each cache being dropped from the logical processor and updating the cache mapping. Dirty cache lines in the remaining cores are written back only when a cache line is evicted.

2.2 Speculation

It has long been recognized that speculation is an essential piece of achieving good performance on serial workloads. E2 makes aggressive use of speculation to improve performance. A combined predicate-branch predictor [5] speculates at two levels. First, it predicts the branch exit address for each block for speculation across blocks. Second, it predicts the control flow path within blocks by predicting the predicate values.

2.2.1 Speculation Across Blocks

Predicting the branch exit address allows instruction blocks to be fetched and begin executing before the current block has completed. The oldest instruction block is marked as non-speculative, and predicts a branch exit address. This address is fetched and begins executing on another physical core in the logical processor or on the same physical core if there is available space in the instruction window.

The taken branch address often resolves before the block completes. In this case, the non-speculative block notifies the other cores in the logical processor of the taken address.

Component	Parameters	Area (mm^2)	% Area
Instruction Window	32x54b	0.08	2%
Branch Predictor		0.12	3%
Operand Buffers	32x64b	0.19	5%
ALUs	4 SIMD, Int+FP	0.77	20%
Register File	64 x 64b	0.08	2%
Load-Store Queue		0.19	5%
L1 I-Cache	32kB	1.08	28%
L1 D-Cache	32kB	1.08	28%
Control		0.19	5%
Core		3.87	100%
L2 Cache	4MB	100	

Table 1: E2 core components, design parameters, and area.

The oldest instruction block then becomes the non-speculative block. Any blocks that were not correctly speculated are squashed. This taken branch signal differs from the commit signal. The taken branch allows the next block to continue speculation and begin fetching new instruction blocks. However, register and memory values are not valid until after the commit signal.

2.2.2 Speculation Within a Block

There are three types of speculation within an instruction block. Predicate speculation uses the combined predicate-branch predictor to predict the value of predicates. Memory speculation occurs in speculative blocks when the speculative block loads values from the L1 cache that may be changed by less-speculative blocks. Load speculation occurs when the load-store queue (LSQ) allows loads to execute before stores with lower load-store identifiers have executed.

In all three cases, mis-speculation requires re-execution of the entire instruction block. This is relatively lightweight and only requires invalidating the valid bits in all of the operand buffers, and re-loading the zero operand instructions.

2.3 Area and Frequency

We developed an area model for the E2 processor using ChipEstimate InCyte [4] and an industry-average 65nm process technology library. The design parameters and component areas are shown in Table 1. Each E2 core requires 3.87 mm^2, including L1 caches.

Frequency estimates are not available using our version of InCyte. However, the microarchitecture does not have any large, global structures and uses distributed control throughout the chip. Because of this, we expect that E2 will achieve a comparable frequency to standard cell ARM multi-core processor designs, which range from 600 to 1000 MHz in 65nm [2].

3. EXECUTION PIPELINE

E2's execution is broken into three primary stages: instruction fetch, execute, and commit. This section first describes the behavior of each stage when operating in scalar mode, then describes the differences between scalar and vector mode.

3.1 Fetch

One of the primary differences between E2 and conventional architectures is that E2 fetches many instructions at

once, rather than continually fetching single instructions. Instructions Blocks of up to 128 instructions are fetched from the L1 instruction cache at a time and loaded into the instruction window. Instructions remain resident in the window until block commit (or possibly longer, as described in section 3.3.1).

Physical cores support one 128-instruction block, two 64-instruction blocks, or four 32-instruction blocks in the window at the same time. Instruction blocks begin with a 128-bit block header containing: the number of instructions in the block, flags for special block behavior, and bit vectors that encode the global registers written by the block and the store identifiers used. Instructions are 32-bits wide, and generally contain at least four fields:

- *Opcode* [9 bits]: The instruction to execute along with the number of input operands to receive.
- *Predicate* [2 bits]: Indicates whether the instruction must wait on a predicate bit, and whether to execute if that bit is true or false.
- *Target 1* [9 bits]: The identifier of the consumer for the instruction's result. If the consumer is a register, this field is the register number. If the consumer is another instruction, this field contains the consumer's instruction number (used as an index into the operand buffer) and whether the result is used as operand 0, operand 1, or as a predicate.
- *Target 2 / Immediate* [9 bits]: Either a second instruction target, or a constant value for immediate instructions.

The instruction window is divided into four equal banks with each bank loading two instructions per cycle. Instructions that do not require input operands, such as constant generation instructions, are scheduled to execute immediately by pushing the instruction number onto the ready queue.

3.2 Execute

Execution starts by reading ready instruction numbers from the ready queues. Operands, the opcode, and the instruction target fields are forwarded to either the ALUs, register file (for `read` instructions), or the load-store queue (for loads and stores). The target field is used to route the results (if any) back to the appropriate operand buffer (or to the register file, in the case of writes).

When results are forwarded back to the operand buffers, the targeted instruction is checked to see which inputs are required, and which operands have already arrived. If all operands for the instruction have arrived, the instruction number is added to the ready queue. Execution continues in this data-driven manner until the block is complete.

Like other EDGE and dataflow architectures, special handling is required for loads and stores to ensure that memory operations follow the program order semantics of imperative language programs. E2 uses the approach described in [10], where the compiler encodes each memory operation with a sequence identifier to denote program order that the microarchitecture uses to enforce sequential memory semantics.

Not all instructions in a block necessarily execute because of predication, so the microarchitecture must detect block completion. Blocks are considered complete when (1) one (and only one) branch has executed, and (2) all instructions that modify external state (register writes and stores) have executed. The compiler encodes the register writes and store identifiers in the instruction block header so that the microarchitecture can identify when criteria (2) is satisfied.

Element Size	Minimum (1 ALU)	Maximum (4 ALUs)
8-bit	8	32
16-bit	4	16
32-bit	2 (1 single fp)	8 (4 single fp)
64-bit	1	4

Table 2: Supported vector operations

3.3 Commit

During execution, instructions do not modify the architectural state. Instead, all changes are buffered and commit together at block completion. Once the core enters the commit phase, the register file is updated with all register writes, and all stores in the load-store queue are sent to the L1 cache beginning with the lowest sequence identifier. Once all register writes and stores have committed, the core sends a commit signal to all other cores in the same logical processor.

3.3.1 Refresh

One important commit optimization, called *refresh*, occurs when the instruction block branches back to itself. Rather than loading the instructions again from the L1 instruction cache, the instructions are left in place and only the valid bits in the operand buffers and load-store queues are cleared. This allows the instruction fetch phase to be bypassed entirely. Instructions that generate constants can also pin values in the operand buffers so that they remain valid after refresh, and are not regenerated each time the instruction block executes.

3.4 Vector Mode

Vector mode execution divides each processor core into N (in this paper 4) independent vector lanes. When operating in vector mode, instructions can only target other instructions in the same vector lane, eliminating the need for a full crossbar between operand buffers and ALUs. Each lane consists of a 32-entry instruction window, two 64-bit operand buffers, 16 registers, and one ALU.

E2 supports vector operations on 64-bit, 128-bit (padded to 256-bits), and 256-bit wide vectors. Each ALU supports eight 8-bit, four 16-bit, or two 32-bit vector operations. Four ALUs enable E2 to support up to 32 vector operations per cycle per core. 64-bit vector operations utilize a single ALU, where as 128- and 256-bit vector operations utilize all four ALUs. Table 1 lists the number of parallel vector operations supported for each vector length and data element size.

Instruction blocks containing vector instructions are limited to 32 instructions which is the size of the instruction window for each vector lane. Vector instructions issuing in lane 1 are automatically issued in the other three lanes and scalar instructions are always assigned to lane 1.

In vector mode, the sixty-four 64-bit physical registers (R0 - R63) are aliased to form sixteen 256-bit vector registers (V0 - V15). We divide the physical register file into four banks to support single cycle access for vectors.

3.4.1 Memory Access in Vector Mode

E2 cores operate on vectors in 256-bit chunks, which enables efficient exploitation of data-level parallelism on small and medium-length vectors. Operating on larger vectors is

done using multiple instruction blocks in a loop, using the efficient refresh mode to bypass instruction fetch and the generation of constants (section 3.3).

Splitting larger vectors among multiple instruction blocks could introduce a delay between loads of adjacent chunks of the same vector, as those loads are split among multiple instruction blocks. To mitigate this delay, E2 employs a specialized unit called the *memory interface controller* (MIC). The MIC takes over control of the L1 data cache, changing part of the cache into a prefetching stream buffer [8, 11]. Stream buffers predict the address of the next vector load and bring that data into the cache early. This ensures that the vector loads in subsequent instruction blocks always hit in the L1 cache.

Since vector and scalar operations are mixed in instruction blocks, part of the cache still needs to operate as a traditional cache. Rather than halve the size of the cache, the set associativity of the cache is cut in half – converting those ways into the memory for a stream buffer. On vector loads, the cache checks the stream buffer. On scalar loads and stores, the cache checks the cache in the same manner, albeit with a reduced number of sets to check.

Vector store instructions are buffered in the stream buffer until block commit, at which point they are written directly out to main memory.

4. EXAMPLE: RGB TO Y CONVERSION

In this section we give an example to explain how a program is vectorized on E2. Figure 2 shows the C code and corresponding vectorized assembly for a RGB to Y brightness conversion which is commonly used to convert color images to grayscale. Each pixel in an image has a triple corresponding to the red, green, and blue color components. Brightness (Y) is computed by multiplying each RGB value by a constant and summing the three results. This program can be trivially parallelized to perform multiple conversions in parallel since each conversion is independent.

4.1 C Source

Each RGB component is represented by a vector and pointers to these vectors, along with a pointer to the preallocated result vector Y, and the number of vectors to convert are passed to the function as arguments (lines 4-5). The constants for the conversion are also stored in vectors (lines 7-12). Each vector is 256-bits wide and the individual data elements are padded to 64-bits since their type is a 32-bit single precision float. The conversion is done using a simple for loop (lines 14-16). To simplify the example we do not unroll the loop to fill the block.

4.2 Assembly

The assembly listing is given in lines 18-51. Instructions are grouped into blocks by the compiler (one block for this example) and a new block begins at every label (line 18). The architecture fetches, executes, and commits blocks atomically. By convention we use Rn to denote scalar registers, Vn to denote vector registers, and Tn to denote temporary operands. The scalar and vector registers are part of the global state visible to every block. The temporary operands however are only visible within the block they are defined. The only instructions that can read from the global register file are register READ instructions (lines 19-27). However, most instructions can write to the global register file.

```
1  // numVectors > 0
2  // y = r * .299 + g * .587 + b * .114;
3  void rgb2y(int numVectors,
4    __vector float *r, __vector float *g,
5    __vector float *b, __vector float *y)
6  {
7  __vector float yr = { 0.299f, 0.299f,
8     0.299f, 0.299f };
9  __vector float yg = { 0.587f, 0.587f,
10    0.587f, 0.587f } ;
11 __vector float yb = { 0.114f, 0.114f,
12    0.114f, 0.114f };
13
14 for (int i = 0; i < numVectors; i++)
15   y[i] = r[i] * yr + g[i] * yg + b[i] * yb;
16 }
17
18 _rgb2y:
19   read t30, r3    // numVectors
20   read t20, r4    // address of next r
21   read t21, r5    // address of next g
22   read t22, r6    // address of next b
23   read t32, r7    // address of y
24   read t31, r8    // i
25   read t1, v0     // vector yr
26   read t3, v1     // vector yg
27   read t5, v2     // vector yb
28
29 // RGB to Y conversion
30   vl t0, t20 [0]       // vector load
31   vl t2, t21 [1]
32   vl t4, t22 [2]
33   vfmul t6, t0, t1     // vector fp mul
34   vfmul t7, t2, t3
35   vfmul t8, t4, t5
36   vfadd t9, t6, t7     // vector fp add
37   vfadd t10, t8, t9
38
39 // store result in Y
40   multi t40, t31, #32
41   add t41, t32, t40
42   vs 0(t41), t10 [3]  // vector store
43
44 // loop test
45   tlt t14, t31, t30
46   ret_t<t14>
47   br_f<t14> rgb2y
48   addi r8, t31, #1
49   addi r4, t20, #32
50   addi r5, t21, #32
51   addi r6, t22, #32
```

Figure 2: C source code and E2 assembly listing for a vectorized RGB to Y brightness conversion.

Vector instructions begin with 'v' (lines 30-37 and 42). All load and store instructions are assigned load-store identifiers to ensure sequential memory semantics (lines 30-32 and 42). That is a load assigned ID0 must complete before a store with ID1.

Most instructions can be predicated and predicates are only visible within the block they are defined. Predicated instructions take an operand representing true or false that is compared against the polarity encoded into the predicated instruction (denoted by _t and _f). The test instruction in line 45 creates a predicate that the receiving instructions (lines 46-47) compare against their own encoded predicate. Only instructions with matching predicates execute.

CYCLE	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
FETCH	IF	IF	IF	IF													
READ		R	R	R	R	R											
READ		R	R	R	R												
MEM			L	L	L												S
EX			A	A	A	M	M	M	M	M	M	M	M	M	A	A	B
EX			M	A	T	M	M	M	M	M	M	M	M	M	A	A	A
EX						M	M	M	M	M	M	M	M	M	A	A	
EX						M	M	M	M	M	M	M	M	M	A	A	

Figure 3: One possible schedule for Figure 2.

Blocks are limited to a maximum of 128 scalar instructions. When using vector instructions blocks are limited to a total of 32 scalar and vector instructions. Block _rgb2y contains a mix of 27 scalar and vector instructions.

4.3 Instruction Schedule

Figure 3 gives one possible schedule for the example in Figure 2. We assume a three cycle 32-bit floating point multiply, and that all loads hit in the L1 cache and require three cycles. The architecture is capable of fetching eight instructions per cycle and thus requires four cycles to fetch the 27 instruction block. In cycle one, eight register read instructions are fetched which are all available for execution in the following cycle since they have no dependencies. Two register reads can execute per cycle requiring five cycles to read all the global registers. In cycle two, registers R4 (line 20) and R8 (line 24) are read and the resulting data is sent to the vector load (line 30), multiply immediate (line 40), and add immediate (line 48) instructions. Since each of these instructions is waiting on a single operand, they are now all ready and begin executing in cycle three. Execution continues in this dataflow fashion until the block is ready to commit in cycle 17.

5. CONCLUSION

In this paper we described the E2 architecture – a new dynamic multicore utilizing an Explicit Data Graph Execution (EDGE) ISA, designed to achieve high performance power efficiently. As an EDGE architecture, E2 efficiently exploits instruction-level parallelism through dataflow execution and aggressive speculation. In addition, we have described how the architecture adapts to handle data-level parallelism through vector and SIMD support. This vector support can be interspersed with scalar instructions, making E2 more flexible than traditional vector processors, and more capable than traditional scalar architectures.

We have developed an architectural simulator for E2 using SystemC and a new compiler backend with the Microsoft Phoenix software optimization and analysis framework [1]. We are currently developing a cycle accurate FPGA implementation that when combined with our industrial strength compiler, will allow us to perform a detailed exploration and evaluation of the architecture.

Many challenges lay ahead. To be compelling as an accelerator, we must demonstrate that E2 provides better performance, power efficiency, and programmability than specialized accelerators such as GPUs and dedicated vector processors. E2 may also excel as a general-purpose processor, in which case we must show that it provides compelling enough power/performance gains over current static multicore architectures to justify a transition to a new ISA.

E2's performance and power efficiency are built on its ability to compose and decompose cores dynamically, so the correct policies and mechanisms for managing dynamic composition will require careful consideration. Ideally we would like to leave all decisions about composition to the runtime system, freeing the programmer completely from reasoning about the underlying hardware.

Finally, there is a wide variety of application domains where E2's ability to trade-off power and performance could be useful, ranging from embedded devices to the data center. In the months ahead we plan to examine a diverse set of workloads and investigate how broadly an E2 processor can span the power-performance spectrum.

6. REFERENCES

[1] Microsoft Phoenix. http://research.microsoft.com/phoenix/.

[2] ARM. Cortex-A9 MPCore technical reference manual, November 2009.

[3] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and the TRIPS Team. Scaling to the End of Silicon with EDGE architectures. *IEEE Computer*, 37(7):44–55, July 2004.

[4] Cadence. Cadence InCyte Chip Estimator, September 2009.

[5] H. Esmaeilzadeh and D. Burger. Hierarchical control prediction: Support for aggressive predication. In *Proceedings of the 2009 Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures*, 2009.

[6] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *IEEE COMPUTER*, 2008.

[7] E. İpek, M. Kırman, N. Kırman, and J. F. Martínez. Core Fusion: Accommodating software diversity in chip multiprocessors. In *International Symposium on Computer Architecture (ISCA)*, San Diego, CA, June 2007.

[8] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *SIGARCH Computer Architecture News*, 18(3a), 1990.

[9] C. Kim, S. Sethumadhavan, D. Gulati, D. Burger, M. Govindan, N. Ranganathan, and S. Keckler. Composable lightweight processors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[10] S. Sethumadhavan, F. Roesner, J. S. Emer, D. Burger, and S. W. Keckler. Late-binding: enabling unordered load-store queues. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 347–357, New York, NY, USA, 2007. ACM.

[11] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, 2000.

[12] A. Smith. *Explicit Data Graph Compilation*. PhD thesis, The University of Texas at Austin, 2009.