Task Superscalar: An Out-of-Order Task Pipeline

Yoav Etsion[*] Felipe Cabarcas[*‡] Alejandro Rico[*] Alex Ramirez[*†] Rosa M. Badia[*§]
Eduard Ayguade[*†] Jesus Labarta[*†] Mateo Valero[*†]

[*]*Barcelona Supercomputing Center (BSC)*
Barcelona, Spain

[†]*Universitat Politècnica de Catalunya (UPC)*
Barcelona, Spain

[‡]*Universidad de Antioquia*
Medellín, Colombia

[§]*Artificial Intelligence Research Institute (IIIA)*
Spanish National Research Council (CSIC)

***Abstract*—We present *Task Superscalar*, an abstraction of instruction-level out-of-order pipeline that operates at the task-level. Like ILP pipelines, which uncover parallelism in a sequential instruction stream, task superscalar uncovers task-level parallelism among tasks generated by a sequential thread. Utilizing intuitive programmer annotations of task inputs and outputs, the task superscalar pipeline dynamically detects inter-task data dependencies, identifies task-level parallelism, and executes tasks out-of-order.**

Furthermore, we propose a design for a distributed task superscalar pipeline frontend, that can be embedded into any manycore fabric, and manages cores as functional units.

We show that our proposed mechanism is capable of driving hundreds of cores simultaneously with non-speculative tasks, which allows our pipeline to sustain work windows consisting of tens of thousands of tasks. We further show that our pipeline can maintain a decode rate faster than 60ns per task and dynamically uncover data dependencies among as many as ∼50,000 in-flight tasks, using 7MB of on-chip eDRAM storage. This configuration achieves speedups of 95–255x (average 183x) over sequential execution for nine scientific benchmarks, running on a simulated CMP with 256 cores.

Task superscalar thus enables programmers to exploit many-core systems effectively, while simultaneously simplifying their programming model.

***Keywords*-Out-of-order execution, CMP/manycore, task superscalar, parallel programming**

I. Introduction

The move to on-chip parallelism has motivated research into simplified parallel programming models. Now, we witness a growing popularity of task-based programming models such as Cilk [3], OpenMP 3.0, Intel TBB, CUDA, and OpenCL, suggesting that the task abstraction is an intuitive programming construct.

But a major drawback of common task-based models is burdening the programmer with the non-trivial assignment of resolving inter-task data dependencies. Moreover, many task-based models utilize (possibly partial) barrier synchronization, which inhibits utilization of distant parallelism. Figure 1 illustrates both problems. It depicts the dependency graph for a Cholesky decomposition of a 5x5 matrix, showing that even a small input set may result in an irregular dependency graph, which accommodates distant parallelism — for example, the graph shows that the 6th and 23rd tasks (of 35) can, in fact, run in parallel.

An emerging class of task-based dataflow programming models automates data dependency resolution. These models use programmer annotations of input and output operands to kernel functions[1]to dynamically construct the inter-task data dependency graph, and extract task parallelism at runtime. Such models include Jade [20], StarSs [2], [17], Intel RapidMind [14], OoOJava [10], and Sequoia [7]. However, these models rely on software-based dependency analysis, which is inherently slow, and impedes their scalability [19].

In contrast, traditional out-of-order pipelines excel in fast decoding of inter-instruction data dependencies. Adapting out-of-order pipelines to operate at the task-level will provide a methodology to effectively utilize large manycore fabrics, as well as greatly simplify their programming.

In this paper, we present task superscalar multiprocessors, a task-level abstraction of dynamically scheduled out-of-order processors that manages cores as functional units. By employing task-based dataflow programming constructs, task superscalar multiprocessors identify inter-task dependencies, construct the data dependency graph, and schedule tasks for execution. Task superscalar multiprocessors, therefore, provide programmers with the same abstraction that makes out-of-order processors so appealing: a seemingly sequential interface for a parallel execution engine.

To demonstrate the effectiveness of task superscalar multiprocessors, this paper also presents a design for a distributed task superscalar pipeline frontend, which can be embedded into virtually any manycore fabric, and manage it as a task superscalar multiprocessor. We show that the dependency decode rate provided by the proposed design, enables driving large manycores with relatively fine-grain tasks.

The high-level operational flow of task superscalar is illustrated in Figure 2. A task-generating thread sends tasks to the

[1]Throughout this paper, we use the term *task* to refer to a dynamic instance created when invoking specially annotated *kernel* functions (which, in general, can also consist of independent code blocks [10]).

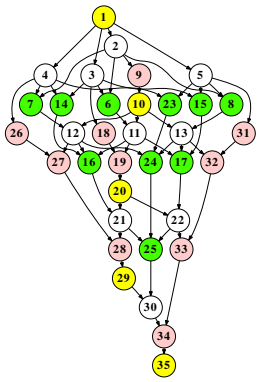

Figure 1. Task graph for a 5x5 Cholesky decomposition. The numbers indicate the task creation order, and the shades represent different kernels.

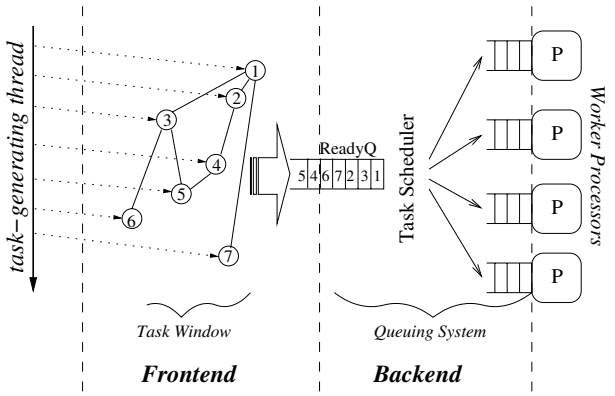

Figure 2. High-level view of Task Superscalar.

pipeline frontend for dependency decoding. However, as the execution of the task-generating thread is decoupled from that of the tasks themselves, the inter-task control path is resolved by the task-generating thread, and the tasks received by the pipeline are *non-speculative*. The pipeline frontend maintains a window of recently generated tasks, for which it generates the data dependency graph, and uncovers task-level parallelism. Importantly, as tasks are non-speculative, the task window can consist of tens-of-thousands of tasks, which enables it to uncover large amounts of parallelism [6]. Furthermore, the pipeline increases available parallelism by renaming memory objects, thus breaking anti- and output-dependencies. Finally, ready tasks are sent to the execution backend, which consists of a task scheduler, a queuing system, and a manycore fabric. Still, the backend can also function as a regular chip multiprocessor (CMP).

In addition, through its inherent support of task-based dataflow execution, task superscalar facilitates high-level programming paradigms, such as *MapReduce*, Intel *Ct* [8], and Intel *CnC* [12].

We evaluate the scalability of the task superscalar pipeline, and explore the effect of its distributed design on the decode rate of data dependencies. In addition, we evaluate the effectiveness of the task window size on the amount of parallelism uncovered and the resulting speedups. Finally, we compare the pipeline's scalability to that of the highly tuned software dependency decoder of the StarSs programming model. Our evaluation workload consists of 9 scientific applications and kernels (listed in Table I), that were parallelized using the StarSs programming model.

The paper is organized as follows: the following section motivates fast decoding of data dependencies as the key to the scalability of task-based dataflow models. Next, Section III discusses the implications of applying ILP techniques at the task-level, and presents the StarSs programming model. Section IV presents the design of the task superscalar pipeline. Our experimental methodology is discussed in Section V, and the evaluation is presented in Section VI. Finally, Section VII discusses related work, and we conclude in Section VIII.

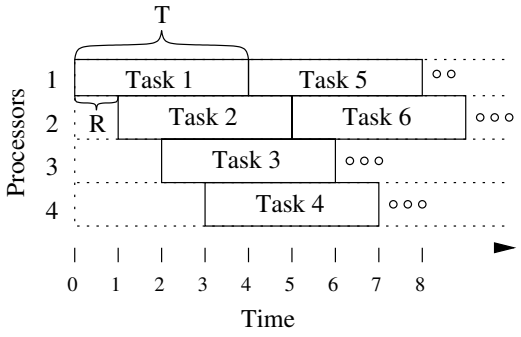

Figure 3. The maximal amortized decode latency allowed in order to maintain best machine utilization: given a fixed task of runtime T, and a machine with P processors, the decode rate R must not exceed $\frac{T}{P}$.

II. Motivation: Fast Dependency Decode Rate

Following the creation of a task, its data dependencies must be identified, so it can be added to the task graph. We refer to this as *task decode*. Effective utilization of a CMP thus requires that tasks be decoded faster than they are consumed by the processing units. Fast decoding of tasks is thus essential for utilizing large CMPs.

Figure 3 illustrates this balance by depicting an ideal scheduling of tasks with runtime T, on a machine with $P = 4$ processors. To maintain good utilization, P tasks must be dispatched every T time units, or one task every $R = \frac{T}{P}$. Correspondingly, this is the target decode rate. Scaling the number of processors P thus requires either increasing the decode rate, or decreasing the execution rate by programming longer tasks.

Increasing task runtimes, however, will degrade overall system performance as the computation will quickly become memory bound: as the runtime of a given code, running on

a given processor, is determined by the size of its input dataset (through a relation approximated by its algorithmic computational complexity), longer runtimes mandate larger datasets. However, once the dataset exceeds the capacity of the per-core L1 cache, the code will start suffering from memory stalls, and performance will degrade.

The application set, listed in Table I, was therefore optimized (when possible) for L1-sized 64KB blocks. The table lists the task runtimes and memory usage for the selected application set, as measured on the simulation platform. For example, MatMul tasks process 48KB of data in $23\mu s$. Therefore, for $P = 256$, a new MatMul task must be decoded every $R = 90\ ns$.

Furthermore, the target decode rate is not determined by the average task runtime, but rather by runtime of the shortest tasks of an application — which are the first to affect the overall utilization. Therefore, given that the shortest tasks of all benchmarks average at $15\mu s$, the rule of thumb suggests a pipeline targeting a 256-way CMP should maintain decode rate in the order of $\frac{15\mu s}{256} = 58\ ns/task$.

But such decode rates are more than an order of magnitude faster than what is achievable in software. As a baseline, we have measured the average decode rate for the highly tuned decoder of StarSs, to be just over 700ns, running on a 2.66GHz Intel Core Duo. Moreover, Rico et al. reported a rate of $\sim 2.5\mu s$ for the Cell BE version of same decoder [19].

This gap between the required decode rate, and that achievable in software, is the focus of this paper.

III. Tasks as Abstract Instructions

Traditional out-of-order pipelines provide programmers with a sequential interface, yet internally execute instructions in parallel, based on dynamic analysis of data dependencies [16]. This section, therefore, discusses how dynamic dependency analysis can be extended to operate at the task-level.

Dynamic identification of data dependencies in out-of-order processors operates by matching each input register of a newly fetched instruction (data consumer), with the most recent instruction that writes data to that register (data producer). The instruction is then sent to a reservation station to wait until all its operands are ready for execution. The reservation stations thus effectively store the instruction dependency graph, which consists of all in-flight instructions.

Processors supporting register renaming, typically match consumers to producers through a lookup in the register renaming table, which maps an architected register name to its assigned physical register. By remapping architected register names to free physical registers, such processors break anti- and output-dependencies (WaR and WaW). Renaming a register thus creates a new version of an architected register. Therefore, register files effectively maintain multiple versions of architected registers.

The matching of data consumers to producers thus relies on two requirements:

- The instruction/task decode mechanism must identify all possible effects an instruction might have on the shared processor state. This requirement enables the decode mechanism to correctly identify data producers and consumers, and maintain a consistent view of the different data versions.
- Instructions/tasks are decoded in-order[2]. This requirement guarantees correct ordering of data producers and consumers, and specifically, that the decoding of an instruction producing a datum updates the renaming table, before any instruction consuming the datum performs a table lookup.

Applying these two principles to tasks, enables the design of an equivalent dynamic decode mechanism that tracks data dependencies at the task-level.

A. Exposing Task Effects on Shared State

Instructions interact with the shared state of a processor, either explicitly or implicitly. Explicit interactions are specified as the source and destination register operands (we simplify the discussion by assuming a load/store ISA), whereas implicit interactions typically consist of accessing the processor status register (reading/writing condition codes, for example). But since all instructions, their operand directionality, and side-effects, are part of the ISA definition, this information is encoded into the processor design.

Applying these principles to tasks, however, implies that all their interactions with shared state must be explicitly exposed as task operands, whose directionality — whether it is an input or an output operands — must be explicitly specified. Therefore, tasks may not have hidden side-effects.

Unlike instruction operands, which consist of architected register names and immediate values, task operands consist of memory objects and scalar values. The pipeline must therefore identify data dependencies among memory objects. While said objects need not be consecutive, our analysis is currently limited to consecutive memory, in order to simplify the pipeline design.

Memory operands are therefore represented as tuples consisting of *operand type*, *base pointer*, *object size*, and *directionality*. The type field indicates whether the operand is a memory object or a scalar value, and the directionality indicates whether the operand is an input operand, output operand, or both. Scalars are equivalent to immediate values, and can only be used as inputs.

Exposing the task operands is therefore a static operation, and is either performed by the programmer when writing the kernel function, or deduced by the compiler. Although seemingly restrictive, the model is in fact quite powerful, as described below.

[2]This definition also covers speculative instruction fetch, as it is the instruction *path* that is speculated, not the order of instructions in the path.

Table I
THE BENCHMARK APPLICATIONS USED IN THE PAPER, AND THEIR RESPECTIVE TASK INFORMATION, MEASURED ON THE SIMULATION PLATFORM.

Name	Class	Description	Task Information				Decode Rate[1]
			Data Sz.	Runtime (μs)			
			Avg.	Min	Med	Avg	(ns/task)
Cholesky	Math. kernel	Blocked Cholesky decomposition	47 KB	16	33	31	63
MatMul	Math. kernel	Blocked matrix multiplication	48 KB	23	23	23	90
FFT	Signal Processing	2D Fast Fourier Transform	10 KB	13	14	26	51
H264	Multimedia	Decoding a HD clip	97 KB	2	115	130	8
KMeans	Machine Learning	K-Means clustering	38 KB	24	59	55	94
Knn	Pattern Recognition	K-Nearest Neighbors	10 KB	17	107	109	66
PBPI	Bioinformatics	Bayesian Phylogenetic Inference	32 KB	28	29	29	108
SPECFEM	Physics (Earth)	Seismic wave propagation	770 KB	9	14	49	35
STAP	Physics (Radar)	Space-Time Adaptive Processing	8 KB	1	9	28	4
		Average[2]	110 KB	15	45	53	58

[1]Decode rate limit is calculated for a for a 256-way CMP. [2]The average data size excluding SPECFEM is 32KB.

B. In-order Task Decode

Tasks that are generated by a single thread maintain this property by definition, as the thread itself is sequential. This is in fact analogue to a processor operating on a single instruction stream. The single-threaded case is easily extended to support multiple task-generating threads by partitioning data between threads. Data partitioning maintains the in-order property at the thread level, as tasks emanating from different threads have no data dependencies. Still, as this paper focuses on the *concept* of task superscalar pipelines, we limit the discussion the single-threaded case.

C. The StarSs Programming Model

The StarSs programming model [2], [17] supports out-of-order execution of tasks, by enabling programmers to explicitly expose task side-effects, through annotating operands of kernel functions as *input*, *output*, or *inout* (bidirectional). The model can thus decouple the execution of the thread generating the tasks, from their decoding and execution.

Figure 4 shows an example of a blocked Cholesky matrix decomposition, programmed with StarSs. The top of the figure shows the StarSs annotations of the kernels, describing the directionality of each kernel operand. The task-generating code itself, shown on the bottom, is identical to a sequential implementation of the algorithm.

At runtime, whenever the task-generating thread reaches a call site to one of the kernels, task creation code (injected by a source-to-source compiler) packs the kernel code pointer and all the task operand values, and writes the data to the task pipeline. As the execution of the task-generating thread is decoupled from the execution of the tasks, it can then resume execution, and continue to spawn additional tasks (the thread is only stalled when the task window becomes). The pipeline, on the other hand, asynchronously decodes the task dependencies, generates the data dependency graph, and schedules tasks as they become ready.

```
#pragma css task input(a, b) inout(c)
void sgemm_t(float a[M][M], float b[M][M],
             float c[M][M]);

#pragma css task inout(a)
void spotrf_t(float a[M][M]);

#pragma css task input(a) inout(b)
void strsm_t(float a[M][M], float b[M][M]);

#pragma css task input(a) inout(b)
void ssyrk_t(float a[M][M], float b[M][M]);
```

```
float A[N][N][M][M]; // NxN blocked matrix,
                     // with MxM blocks
for (int j = 0; j<N; j++) {
   for (int k = 0; k<j; k++)
      for (int i = j+1; i<N; i++)
         sgemm_t(A[i][k], A[j][k], A[i][j]);

   for (int i = 0; i<j; i++)
      ssyrk_t(A[j][i], A[j][j]);

   spotrf_t(A[j][j]);

   for (int i = j+1; i<N; i++)
      strsm_t(A[j][j], A[i][j]);
}
```

Figure 4. Example of a blocked Cholesky decomposition, programmed with StarSs. Declarations of kernel functions (top) define the building blocks for the task-generating thread (bottom). Note the task-generating code itself is sequential, and the task dependencies graph, facilitating parallelization, is constructed at runtime using only the operand annotations. The kernels themselves comprise of routines from the stock BLAS library, operating on memory blocks.

Such a dependency graph, generated for a Cholesky decomposition of a 5x5 matrix is presented in Figure 1. The graph's highly irregular structure, demonstrates the expressive power of StarSs.

The sequential code flow of StarSs, together with the

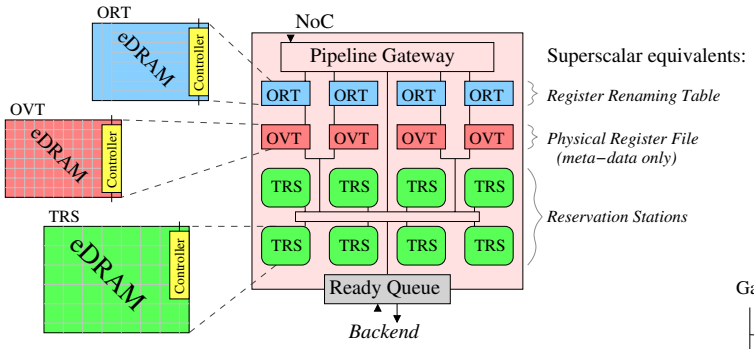

Figure 5. The task superscalar frontend, and organization of the different module types.

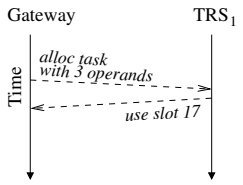

Figure 6. Task allocation.

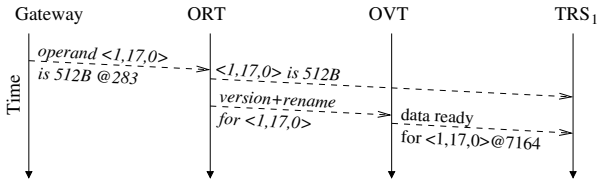

Figure 7. Decoding an **output** operand.

implicit synchronizations and data transfers provided by the runtime, guarantees that the results of a parallel execution will be equivalent to a sequential one.

IV. Pipeline Frontend Design

We describe the frontend top-down, discussing its high-level organization, its operational flow, and key issues in the design of the its constituent modules. As depicted in Figure 5, the frontend employs a tiled design, and is managed by an asynchronous point-to-point protocol. The frontend is composed of four module types:

- A *Pipeline gateway* controls the flow of tasks into the pipeline.
- *Task reservation stations* (TRS), store the in-flight task information and track the readiness of task operands. As such, TRSs are effectively embedded with the data dependency graph. Inter-TRSs communication is used to register consumers with producers, and notify consumers when data is ready.
- *Object renaming tables* (ORT), map operands to their latest version and data producer.
- *Object versioning tables* (OVT), track live operand versions, created whenever a new data producer is decoded. Each OVT is associated with exactly one ORT. The functionality of the OVTs therefore resembles that of a physical register file (although unlike a register file, they only maintain operand *meta-data*). Furthermore, the OVTs break anti- and output-dependencies by renaming operands. Temporary operand buffers are allocated from an OS assigned memory space, and are copied back to the original object address using an external DMA engine.

All modules include a controller, which mostly consists of 1–2KB of transient state. Additionally, the ORTs, OVTs, and TRSs, employ 256–768KB eDRAM blocks to store the various task and operand information. Of these, only the ORTs require associative lookups. All other modules are directly addressed, and protocol messages include the location of the queried datum in the destination module.

The overall performance of the pipeline is largely determined by its configuration: the number of ORTs, OVTs, and TRSs, as well as their overall storage capacity. We evaluate these tradeoffs in Section VI.

A. Operational Flow

The operational flow of the proposed pipeline is an adaptation of traditional out-of-order pipelines. Tasks are decoded, and stored in the TRS until all their operands are ready, at which time they are sent to the ready queue. Whenever a task finishes, the TRS traverses all the task operands and sends *data ready* messages to all its data consumers. In addition, it notifies the OVTs managing its operands to decrement the usage count of the associated data versions. Finally, it frees the space occupied by the task meta-data.

We now turn to describe key elements of the operational flow in more detail.

Processing of tasks begins by allocating task storage space for task meta-data, as depicted in in Figure 6. The gateway sends an allocation request, consisting of the number of task operands, to one of the TRSs (the number of operands determines the storage size). The TRS replies with the task slot number — the TRS internal address of the allocated space. Each task is thus represented by a unique task ID tuple composed of the TRS index and the slot number. For example, the resulting task ID for the allocation shown in Figure 6 would be $< \mathbf{TRS}, \mathbf{SLOT} >=< \mathbf{1}, \mathbf{17} >$. The task ID is also used to derive unique operand IDs, consisting of the task ID and the operand index. The first operand for task $< \mathbf{1}, \mathbf{17} >$ is therefore $< \mathbf{1}, \mathbf{17}, \mathbf{0} >$.

Once a TRS slot is allocated, the gateway initiates the dependency decoding by distributing operands to the ORTs (the target ORT is extracted from the *hashed* operand address). Scalar operands, which do not require dependency tracking, are sent directly to the allocated TRS.

The ORTs mimic the functionality of the register renaming table. Therefore, the ORTs map formal datum ad-

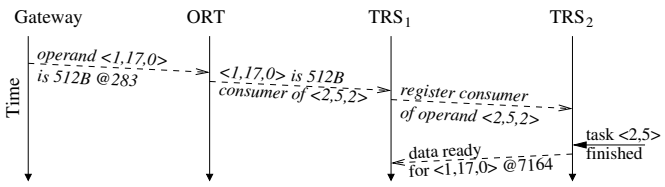

Figure 8. Decoding an **input** operand.

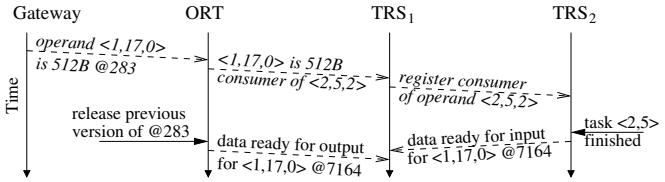

Figure 9. Decoding an **inout** operand.

dresses to operand IDs of data producers (the corresponding output/inout operand of the task generating the data). The exact event flow, however, is determined by the operand directionality.

The decode flows for *output*, *input* and *inout* operands are described in Figures 7, 8, and 9, respectively. When decoding *output* operands, the ORT first sends the basic operand information to the designated TRS. In addition, the ORT initiates an operand renaming request from its associated OVT. The OVT then allocates a new datum version and rename buffer for the operand (analogue to allocating a free physical register, to store the latest version of an architected register, in out-of-order pipelines). Once renamed, the output operand is considered ready, and the OVT sends a *data ready* message to the designated TRS, which includes the address of the rename buffer — memory address 7164 in our example.

Input operands, on the other hand, have to wait for the data producing task to complete, as depicted in Figure 8. The ORT looks up the ID of the last output or inout operand referring to the memory object in question. In the example shown in the figure, the data producer is operand $< \mathbf{2}, \mathbf{5}, \mathbf{2} >$ (operand no. 2 of the task stored in slot 5 in TRS 2). The ORT sends the operand ID to the designated TRS. Upon receiving the operand, the TRS sends a *register consumer* request to the TRS storing the producer task, requesting to be notified when the task finishes. Still, only when the producer task finishes, and a *data ready* message is received, will the operand be considered ready.

The event flow for *inout* operands is a combination of the flows for both input and output operands, as depicted in Figure 9 (note that in this case the operand is not renamed, as it is part of a true dependency). Therefore, the operand needs to receive two *data ready* messages before it is considered ready — one indicating that the input data is in place, and the other indicating that the output buffer is not in use by any reader task. The latter message is received from the OVT after the all tasks using the previous data version finish.

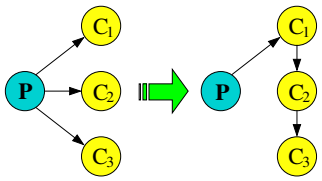

Figure 10. Consumer chaining: all tasks using an operands version are chained, and forward the ready message from one to another.

Although the above description focuses on decoding of individual operands, the pipeline performance stems from its concurrency. As the gateway asynchronously pushes operands to the ORTs, the different decoding flows, task executions, and task terminations, occur in parallel.

B. Module Organization and Design

1) The Pipeline Gateway: The gateway is responsible for allocating TRS space for new tasks, distributing tasks to the different modules, as well as blocking task generating threads whenever the pipeline fills.

New tasks are stored in a dedicated 1KB buffer, which holds over 20 incoming tasks (the exact number depends on the number of operands for each task), and accounts for most of the gateway's size.

For each task, the gateway sends an allocation request to one of the TRSs. As TRSs manage their own storage, the gateway only maintains a queue of TRSs with free space, and selects the first in the queue. Furthermore, the request includes the address of the task in the gateway's internal buffer, which is sent back with the TRS reply, and enables the gateway to directly access the pending task buffer, thus avoiding an associative lookup. Thanks to the non-blocking protocol, the gateway can continue sending allocation requests for newly arrived tasks while waiting for TRS replies.

When the TRS reply is received, the gateway begins issuing task operands to the ORTs, with the designated ORT selected based the operand's base address. However, as memory objects' sizes may vary, basing the ORT selection directly on address bits creates load imbalances, and the address must therefore be hashed. To minimize latencies, hashing is pipelined, and begins as soon as the task arrives at the gateway.

2) Task Reservation Stations (TRS): The TRSs store the meta-data of all in-flight tasks, including the IDs of operands' data consumers, and thereby effectively embed the task dependency graph.

But embedding an irregular structure, such as the task dependency graph, into a distributed memory grid, is non-trivial. Specifically, each task (a graph node) incorporates two degrees of freedom, as both the number of operands

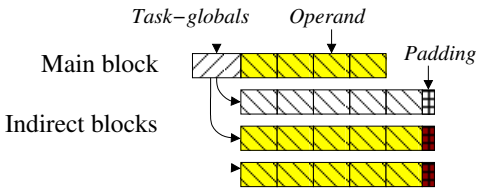

Figure 11. Storing variable-size tasks using fixed-size blocks. The main block stores task-global data first operands. Indirect blocks are used for extra operands.

may vary, as well as the number per-operand consumers. We therefore employ a graph transformation referred to as *consumer chaining*, which eliminates one degree of freedom.

The transformation, illustrated in Figure 10, chains the consumers as a linked list. This requires storing only the operand ID of the first data consumer, instead of a per-operand consumer list. The chaining effectively blurs the roles of producer and consumer, as each consumer serves as its successor's producer. In Figure 10, for example, task C_1 is a consumer from task P's point of view, but a producer for task C_2. When a real producer task finishes, it sends a *data ready* message to the first consumer in the chain, which immediately forwards the message on. Although chaining induces increasingly longer message latencies, we have not witnessed any impact on performance, as chains are typically very short: for all but two of the benchmarks, 95% of the chains are no more than 2 tasks long, and no more than 7 for the other two.

TRSs store task meta-data in a private eDRAM memory, managed as an array of fixed 128B blocks. To accommodate the variable number of operands, we have used a storage layout inspired by UNIX filesystem inodes, as shown in Figure 11. Each task is allocated one main block, which stores the task-global data and the first 4 operands. In addition, the main block may point to three more blocks, each containing up to 5 extra operands, and thereby support up to 19 operands per task. When a task allocation request arrives — which includes the number of task operands — the TRS allocates the required number of blocks, and organizes the task storage space. Although such an allocation method yields internal fragmentation, we have found that the average waste is only $\sim$20% of the allocated memory.

Free blocks are chained as a list, with each node storing 63 pointers to other free blocks, and a pointer to the next node. In addition, the address of the first 64 free blocks are stored in a special 128B SRAM buffer. A typical block allocation is therefore satisfied by the SRAM buffer, and takes only 1 cycle.

Importantly, TRSs do not require associative lookups, as incoming messages contains a task ID tuple, which includes the address of the task's main block.

3) Object Renaming Tables (ORT): The object renaming tables map memory operands to the most recent task accessing the same memory object, and thereby detect object dependencies. Storing any user (either producer or consumer), rather than only storing real data producers, facilitates TRS consumer chaining.

ORT maps, stored in a private eDRAM, are organized as 16-way cache of memory objects, and are looked up using the objects' base address. Tags for each cache set are stored in two 64B blocks, which are read sequentially, and matched against the real operand address. Unlike common caches, however, ORTs never evict entries. If an allocation is requested from a full set, the ORT stalls the gateway until an entry is released. An ORT is therefore a logical cache structure, mapped on top of a standard eDRAM block.

If the evaluated operand read-only, and a match in the cache exists (RaR or RaW), the ORT sends the previous user's operand ID to the designated TRS (as shown in Figures 8 and 9). On the other hand, if the evaluated operand is a writer (output or inout), a new operand version must be created and the OVT is notified. Finally, if no match is found (miss), a new version is created.

4) Object Versioning Tables (OVT): The object versioning tables account for the different live versions of operands. Effectively, the OVT manages data anti- and output-dependencies, either through operand renaming, or by chaining different inout operands and unblocking them in-order (sending a *data ready* message whenever a version is released).

Each OVT entry represents a version of an operand, and mainly includes a usage count (reported by the ORT), a pointer to the next version, and a pointer to the first element in the consumer chain. (the OVT memory management is similar to that of the TRS).

Rename buffer allocation is implemented using a fixed number of buckets, assigned to allocate predetermined power-of-2 sizes. Initially, the operating system assigns each OVT a region in main memory, which is broken into fixed size chunks, and are stored as an in-memory linked list. During allocation, the OVT grabs a buffer from the appropriate bucket, which is refilled with a new memory chunk if empty.

5) Pipeline Backend: The generic CMP substrate includes the execution cores, on-chip network, and cache hierarchy. The pipeline pushes runnable tasks into a queuing system similar to Carbon [13] (although the system currently does not support task stealing). Effectively, processor cores thus serve as functional units.

V. Experimental Methodology

System model: We evaluate task superscalar using TaskSim, a trace-driven cycle-accurate CMP simulator, validated against the Cell BE [18].

Our CMP model, summarized in Table II, consists of 32–256 in-order cores with private L1 caches and a shared L2. Coherence is maintained using a directory-based MSI

Table II
SUMMARY OF THE SIMULATED SYSTEM PARAMETERS.

Cores	32–256 cores, in-order, dual-issue, 3.2GHz
L1	private, 64KB, 4-way set-associative, 3 cycle latency, split D/I
L2	shared, 32 banks with 4MB per bank, 8-way set-associative, 22 cycles latency
Memory	4 memory controllers (MC), 2 channels per MC, single 800MHz DDR3 DIMM per ch.
Interconnect	segmented two-level ring, 16 bytes/cycle, 4 concurrent connections per segment
Task pipeline	22 cycles eDRAM latency, in addition to each module's processing time of 16 cycles

protocol, embedded in the L2. The interconnect is a two-level ring topology. Each core is connected to a processor rings (8 cores per ring), and a global ring connects the processor rings, L2 banks, and the task superscalar frontend.

The task superscalar frontend has an eDRAM access time of 22 cycles. Each pipeline module charges 16 cycles for processing a packet on top of any eDRAM access overheads. Moreover, if packet processing involves multiple operands, the processing overhead is multiplied by the number of operands involved.

Applications: The set of benchmark applications used in our evaluation is listed in Table I. The applications were chosen to represent a broad range of scientific domains, and were ported to the StarSs programming model [17]. The parallelization strategy mainly focused on programmability, attempting to naturally decompose algorithms into tasks. As a result, tasks typically consist of no more than a dozen lines of code.

The StarSs source-to-source compiler replaces calls to kernels with task generation code. The injected code packs kernel information and operand values onto a stack-based memory buffer, which is passed to the software runtime. When interfacing with the task superscalar pipeline, the task generation code maintains the same memory layout, and the buffer is sent to the pipeline.

VI. EVALUATION

The amount of parallelism the task superscalar pipeline can uncover, and its performance, depends on the effective task window size it can sustain. This size not only depends on the amount of physical resources allocated to storing task meta-data, but more importantly, requires that new tasks are decoded and added to the window faster than existing tasks are consumed by the processing units.

We therefore begin our evaluation by exploring the effect of pipeline parallelism (i.e. the number of TRSs, ORTs, and OVTs), on the task decode rate. Following, we evaluate the required storage capacity of the different components, and finally conclude by exploring the scalability of the task superscalar pipeline.

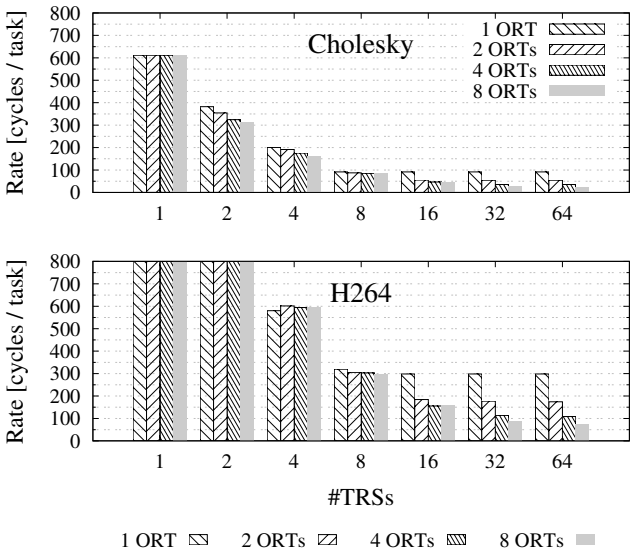

Figure 12. Task decode rate for Cholesky (top) and H264 (bottom).

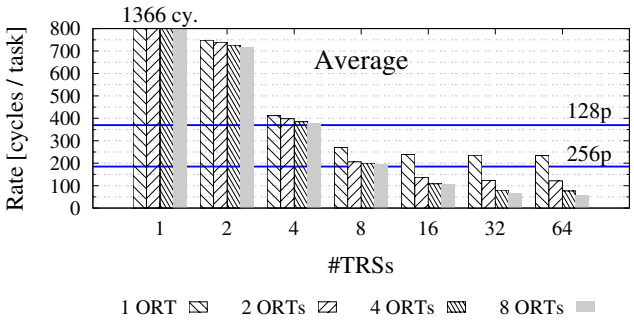

Figure 13. Average task decode rate. The horizontal lines show the rate limits for 128 and 256 processors.

A. Task Decode Rate

The distributed design of the pipeline facilitates speeding up the overall decode rate, by overlapping the decoding of multiple tasks. Specifically, replicating the ORTs (and associated OVTs) enables multiple operands to be decoded in parallel, whereas TRS replication reduces the per-TRS load, and thereby increases the overall processing rate of inter-TRS communication.

Figures 12 and 13 depict the effect of pipeline parallelism on decode rate, measured as the average time between two successive additions to the task graph. The first figure focuses on Cholesky and H264, whereas the second shows the average over all benchmarks. Importantly, both figures show that increasing pipeline parallelism yields consistently faster task decode rates.

Figure 12 demonstrates how the rate varies between benchmarks: with 4 TRSs and 4 ORTs, the average decode rate for Cholesky is less than 185 cycles, or 58ns for our 3.2GHz simulation platform. The same configuration, however, only achieves an average rate of $\sim$300 cycles for

H264. This is caused by the variance in the number of task operands between the two benchmarks: while Cholesky tasks have at most 3 operands, ∼94% of H264 tasks have more than 6. With only 4 ORTs, H264 tasks thus require two ORT iterations, while Cholesky tasks only require one. For the same reason, H264 generates more inter-TRS communication, thereby stressing the TRSs and exerting back-pressure on the ORTs/OVTs. Increasing the number of TRSs reduces their load, alleviates the back-pressure they exert on the ORTs/OVTs, and enables the ORT parallelism to manifest.

Finally, Figure 13 compares the average decode rate with the target rates for 128 and 256 processors, deduced in Section II. Again, it is evident that increasing the pipeline parallelism speeds up the decode rate. Specifically, the figure demonstrates the importance of the distributed embedding of the dependency graph over multiple TRSs: while using multiple ORTs does not affect performance if only a single TRS is present, employing multiple TRSs reduces the task decode rate even if only a single ORT is used. The reason for this effect is that using a single ORT increases the operand decode time, but still allows for concurrent operations on the dependency graph. In contrast, using a single TRS serializes all operations on the task graph.

In conclusion, we determine that using 8 TRSs and 2 ORTs/OVTs is sufficient to support a 256 processor system. We thus use this configuration in our following experiments.

B. Task Window Size

The window size, a product of the aggregate capacity of its constituent modules, poses an important design tradeoff. While large task windows can potentially uncover more parallelism, they also require more on-chip resources. We therefore explore the effects of the task window size on performance, starting with an infinite window, and advancing along the different stages.

ORT and OVT Storage Size: The ORTs maintain an entry for each memory object used by in-flight tasks. As such, their capacity, and the number of operands they can store, affects the number of in-flight tasks.

Figure 14 presents the performance impact of increasing the total capacity of the ORTs. As expected, increasing the ORT capacity facilitates greater speedups, as it increases the task window size, and thereby uncovers more parallelism. The speedups flatten, however, as the ORT capacity reaches a certain level — 128KB for Cholesky, and 512KB for H264 and the average case. At this high-point, the amount of parallelism uncovered by the larger task window increases the rate of task execution, such that it equals the rate in which new tasks arrive at the pipeline. From this point on, performance is no longer limited by the task window, but rather by the task-generating thread which cannot keep up with the increased execution throughput.

This equilibrium is application dependent, and is a combination of the amount of parallelism available, its depth, and the number of operands per task. H264 therefore requires a larger ORT capacity than Cholesky does, both because of its larger number of operands per task, and because it embeds more distant parallelism, which can only be uncovered by a larger task window.

Based on these results, we determine that a total ORT capacity of 512KB provides a good operating point. An equivalent exploration of the OVT design space (not shown), suggests they require a similar capacity.

TRS Storage Size: The TRSs comprise the task window itself. However, it should be emphasized that their storage size does not exclusively determine the effective size of the task window, as the utilization of the task window might be limited by the capacity of earlier pipeline stages — namely the ORTs and OVTs.

Figure 15 therefore shows how increasing the TRS capacities affects the speedups achieved. Once again, the H264 represents the extreme, as the distant parallelism it embeds calls for a total 6MB of TRS memory. Cholesky, on the other hand, has more modest requirements, and peaks at a TRS capacity of 2MB. Finally, on average, the increase is gradual, and while 2MB of TRS memory already provide most of the potential speedup, performance peaks only when the capacity reaches 6MB.

Importantly, a TRS capacity of 6MB provides a the task window of 12,000–50,000 tasks, and is imperative to the pipeline's ability to uncover large amount of parallelism.

C. Task Decode Rate vs. Task Window Size

Task superscalar trades off the size of the task window for fast decoding of data dependencies. In contrast, software-based runtimes provide an effectively infinite window size, that can potentially uncover more parallelism, but are limited in their decode rate.

Figure 16 evaluates this tradeoff, by comparing the scalability of task superscalar with that of the StarSs software runtime. The figure shows that the pipeline scales better than a software runtime for all but one benchmark, and that the software runtime cannot typically utilize more than 32–64 processors.

The only benchmarks for which the software decoder scales up to 128 processors are Knn and H264. The reason is that these two benchmarks consist of relatively long tasks. In both cases, ∼95% of the tasks run for more than $100\mu s$, which implies a rate limit close to 800ns for 128 processors. As a result, the baseline software decode rate of 700ns becomes adequate.

Furthermore, in the case of H264, the software's infinite task window plays a key role. The H264 decoder has a diagonal wavefront parallelism in each frame, and the decoding of each of a frame's macroblocks depends on the decoding

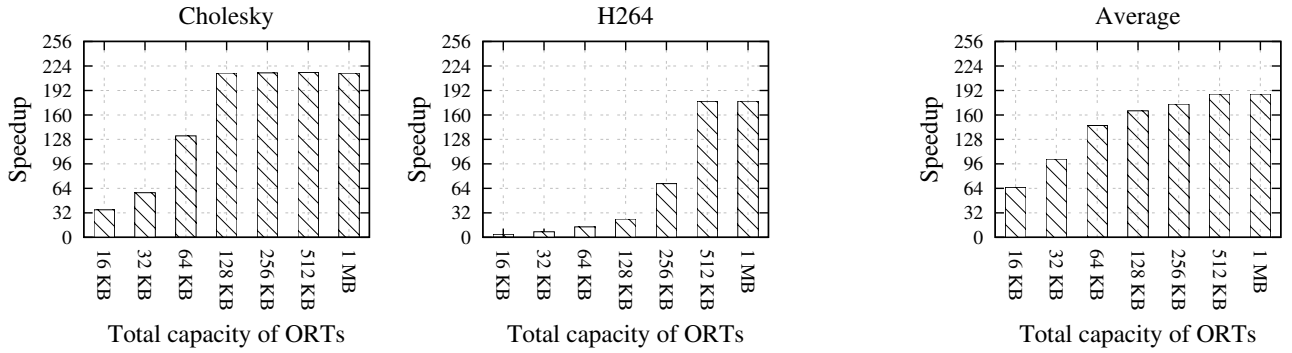

Figure 14. The effect of total ORT size on performance. Results are shown for Cholesky (left), H264 (middle), and the average for all benchmarks (right).

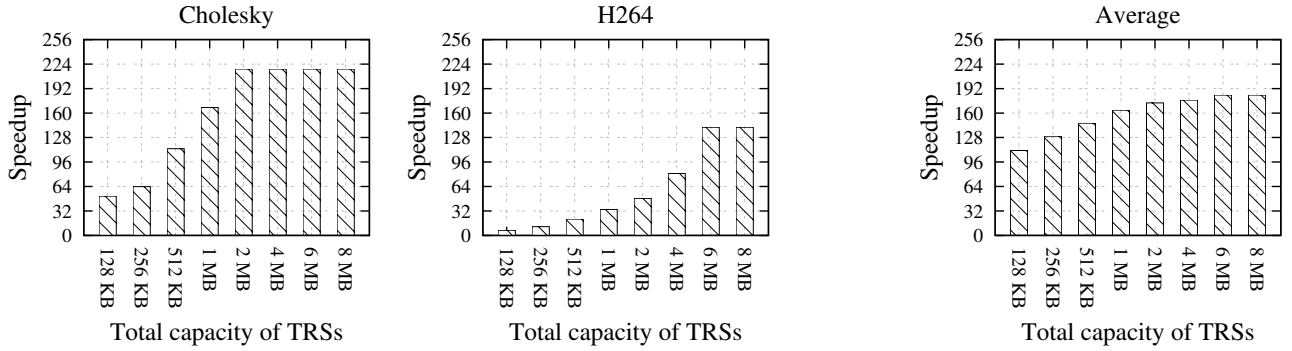

Figure 15. The effect of total TRS size on performance. Results are shown for Cholesky (left), H264 (middle), and the average for all benchmarks (right).

of the macroblocks to its west, north-west, north, and north-east. In addition, each macroblock also depends on the decoding of nearby blocks from its predecessor frame. With over 2000 tasks per frame, and chains of inter-macroblock RaW dependencies that can span up to 60 frames, H264 embeds very distant parallelism which cannot be uncovered by the hardware pipeline. As a result, software's infinite window size prevails, and the software decoder outperforms the hardware pipeline by a small margin.

However, taking advantage of the infinite task window, provided by the software runtime, requires tuning applications for very long tasks. Furthermore, algorithms do not commonly have a computational complexity that requires such long computations for L1-sized inputs. Therefore, increasing task runtime will inevitably require input sets larger than the L1 cache, and might thereby degrade overall system performance by inducing memory stalls.

In conclusion, the fast decode rate provided by the task superscalar pipeline, together with its reasonably large task window, generally outweigh the benefits of the software decoder's infinite task window.

VII. Related Work

The performance scalability of hardware parallelism, alongside the notoriety of explicit parallel programming, pushed the job of uncovering parallelism down to the compiler and hardware, which operate at ILP level. Perhaps most common among ILP designs, are dynamically-scheduled out-of-order processors, which maintain a window of pending instructions, and dynamically schedule them in a dataflow manner [16].

The amount of ILP uncovered by an out-of-order processor directly depends on the size of its instruction window. However, the need for aggressive control-flow speculation, and the quadratic relation between the size of a window and the number of possible dependencies, typically complicate the design of large instruction windows, and limit their effective utilization.

Alternatively, designs such as Multiscalar [24], Trace Processors [21], and the Standford Hydra CMP [9] (to name a few) target coarser parallelism using thread-level speculation (TLS). These designs split a large instruction window into small speculatively independent threads, that can be executed in parallel. The performance benefits of many TLS architectures, however, is impeded by mis-speculated data (in)dependencies.

In contrast, TRIPS [23] and WaveScalar [26], combine both static and dynamic dataflow analysis in order to exploit more parallelism. But these designs, like previous dataflow architectures [1], [5], [15], [28], are still susceptible to memory and communication latencies [4].

The move to parallel architectures, coupled with cumber-

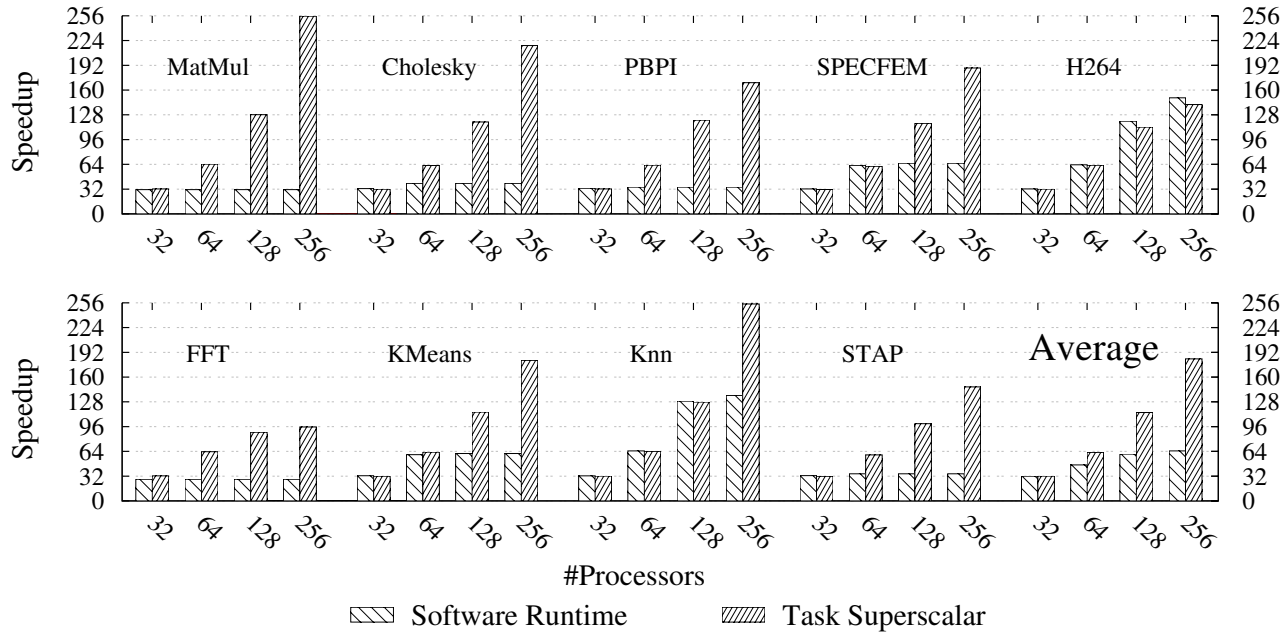

Figure 16. Speedups achieved by the task superscalar pipeline driving 32, 64, 128, and 256 processors, over sequential execution. The speedups are compared with those achieved by a software-based runtime.

someness of existing parallel programming models, raised interest in task-based models. The increasing popularity of models such as Cilk [3], OpenMP 3.0, Intel TBB, CUDA, and OpenCL, attest that tasks provide an intuitive programming construct.

The growing popularity of task-based models has already motivated research into explicit hardware support for tasks. Carbon [13] and ADM [22] use hardware task queues to support fast task dispatch and stealing, whereas the Hyperprocessor [11] manages global dependencies using a universal register file.

However, common task-based models still require programmers to explicitly manage inter-task dependencies, which is far from trivial (OpenCL, for example, supports dynamic scheduling of explicit task graph, but still burdens programmers with manually generating the graph itself). This, in turn, motivated the development of task-based dataflow programming models such as Jade [20], StarSs [2], [17], Intel RapidMind [14], Sequoia [7], OoOJava [10], application specific dataflow backends [25], as well as streaming models such as StreamIt [27]. These models, which are targeted in this paper, take the task abstraction one step further, and use explicit programmer annotations as to the directionality of task operands, in order to dynamically construct the data dependency graph.

Task superscalar, proposed in this paper, generalizes the operational flow of dynamically scheduled out-of-order processors, and provides a native, task-based, dataflow execution engine. Task superscalar therefore combines the effectiveness of out-of-order processors in uncovering parallelism together with the task abstraction, and thereby provides a unified management layer for CMPs — which effectively employs processors as functional units.

VIII. Conclusions and Future Work

We have presented the *Task Superscalar* pipeline, an abstraction of out-of-order superscalar pipelines, that operates at the task-level. Building on an emerging class of programming models that allow programmers to annotate task inputs and outputs, task superscalar facilitates runtime analysis of inter-task data dependencies, and out-of-order task execution. Our experiments show that 7MB of on-chip eDRAM blocks enable the pipeline to store tens-of-thousands of in-flight tasks. Such a large task window enables the pipeline to uncover distant parallelism, and manage large CMPs as a unity.

Furthermore, we show that task superscalar provides fast decoding of data dependencies, and adds new tasks to the task window in less than 60 ns on average. Fast decoding thus enables the pipeline to support fine-grain tasks that execute in whole from the L1 cache, thereby minimizing memory stalls.

Finally, we believe that task superscalar pipelines opens new research directions into managing heterogeneous CMPs at a higher level of abstraction, while leveraging existing knowledge on out-of-order execution.

ACKNOWLEDGMENTS

This research is supported by the Consolider contract number TIN2007-60625 from the Ministry of Science and Innovation of Spain, the TERAFLUX project (ICT-FP7-249013), the ENCORE project (ICT-FP7-248647), the MareIncognito BSC-IBM project, and the European Network of Excellence HIPEAC-2 (ICT-FP7-217068). Y. Etsion is supported by a Juan de la Cierva Fellowship from Ministry of Science and Innovation of Spain. F. Cabarcas is supported in part by the Program AlBan, the European Union Program of High Level Scholarships for Latin America (scholarship No. E05D058240CO).

REFERENCES

[1] Arvind and R. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. on Computers*, 39(3):300–318, Mar 1990.

[2] P. Bellens, J. Perez, R. Badia, and J. Labarta. CellSs: a programming model for the Cell BE architecture. *Supercomputing*, Nov. 2006.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *Symp. on Principles and Practice of Parallel Prog.*, pp. 207–216, 1995.

[4] D. E. Culler, K. E. Schauser, and T. von Eicken. Two fundamental limits on dataflow multiprocessing. In *Intl. Conf. on Parallel Arch. and Compilation Techniques*, pp. 153–164, 1993.

[5] J. B. Dennis and D. Misunas. A preliminary architecture for a basic data flow processor. In *Intl. Symp. on Computer Architecture*, pp. 126–132, 1974.

[6] Y. Etsion, A. Ramirez, R. M. Badia, E. Ayguade, J. Labarta, and M. Valero. Task superscalar: Using processors as functional units. In *Hot Topics in Parallelism*, Jun 2010.

[7] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. *Supercomputing*, 2006.

[8] A. Ghuloum, T. Smith, G. Wu, X. Zhou, J. Fang, P. Guo, B. So, M. Rajagopalan, Y. Chen, and C. B. Future-proof data parallel algorithms and software on intelÂő multi-core architecture. *Intel Technology Journal*, 11(4), Nov 2007.

[9] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, 2000.

[10] J. Jenista, Y. hun Eom , and B. Demsky. OoOJava: An out-of-order approach to parallelizing java. In *Hot Topics in Parallelism*, Jun 2010.

[11] F. Karim, A. Mellan, B. Stramm, A. Nguyen, T. Abdelrahman, and U. Aydonat. The Hyperprocessor: A template System-on-Chip architecture for embedded multimedia applications. In *Workshop on Application Specific Processors*, Dec 2003.

[12] K. Knobe. Ease of use with concurrent collections (CnC). In *Hot Topics in Parallelism*, Mar 2009.

[13] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *Intl. Symp. on Computer Architecture*, pp. 162–173, 2007.

[14] M. McCool. Data-parallel programming on the Cell BE and the GPU using the RapidMind development platform. In *Proc. GSPx Multicore Applications Conf.*, Oct 2006.

[15] G. M. Papadopoulos and K. R. Traub. Multithreading: a revisionist view of dataflow architectures. In *Intl. Symp. on Computer Architecture*, pp. 342–351, 1991.

[16] Y. N. Patt, W. M. Hwu, and M. Shebanow. HPS, a new microarchitecture: rationale and introduction. In *Intl. Symp. on Microarch.*, pp. 103–108, 1985.

[17] J. Perez, R. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Intl. Conf. on Cluster Computing*, pp. 142–151, Sep 2008.

[18] A. Rico, F. Cabarcas, A. Quesada, M. Pavlovic, A. J. Vega, C. Villavieja, Y. Etsion, and A. Ramirez. Scalable simulation of decoupled accelerator architectures. Technical Report UPC-DAC-RR-2010-14, Universitat Politècnica de Catalunya, Jun 2010.

[19] A. Rico, A. Ramirez, and M. Valero. Available task-level parallelism on the Cell B.E. *Scientific Programming*, 17(1–2):59–76, 2009.

[20] M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: A high-level, machine-independent language for parallel programming. *IEEE Computer*, 26:28–38, 1993.

[21] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. *Intl. Symp. on Microarch.*, p. 138, 1997.

[22] D. Sanchez, R. M. Yoo, and C. Kozyrakis. Flexible architectural support for fine-grain scheduling. In *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems*, pp. 311–322, 2010.

[23] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. S. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger. Distributed microarchitectural protocols in the TRIPS prototype processor. In *Intl. Symp. on Microarch.*, pp. 480–491, 2006.

[24] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Intl. Symp. on Computer Architecture*, pp. 414–425, 1995.

[25] F. Song, A. YarKhan, and J. Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *Supercomputing*, pp. 1–11, 2009.

[26] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. WaveScalar. In *Intl. Symp. on Microarch.*, p. 291, 2003.

[27] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *Intl. Conf. on Compiler Construction*, pp. 179–196, Apr 2002.

[28] I. Watson and J. Gurd. A practical data flow computer. *IEEE Computer*, 15(2):51–57, Feb 1982.