Scheduling Directives for Shared-Memory Many-Core Processor Systems

Oded Green*
Computational Science and Engineering
Georgia Institute of Technology
Atlanta, Georgia, USA
ogreen@gatech.edu

Yitzhak Birk
Electrical Engineering
Technion, Israel Institute of Technology
Haifa, Israel
birk@ee.technion.ac.il

Abstract

Abstract— We consider many-core processors with task-oriented programming, whereby scheduling constraints among tasks are decided offline, and are then enforced by the runtime system. Here, exposing and beneficially exploiting fine grain data and control parallelism is increasingly important. Therefore, high expressive power for stating such constraints/directives, along with the ability to implement them in fast, simple hardware, is critical for success. In this paper, we focus on the relationship between duplicable tasks, which are used to express and exploit data parallelism. We extend the conventional Start-After-Complete (precedence) constraint to also be usable between replicas of different such tasks rather than only between entire tasks, thereby increasing the exposable parallelism. Additionally, we propose the parameterized Start-After-Start constraint, which can be used to control the degree of "lockstep" among multiple such tasks, e.g., in order to improve cache performance when the tasks work on the same data. Also, we briefly describe several additional interesting directives. Finally, we show that the directives can be supported efficiently in hardware. Hypercore, a very efficient CREW PRAM-like shared-cache architecture, which is very challenging because it has extremely fast dispatching for basic constraints, is used in the discussion. However, the new directives have broader applicability.

Categories and Subject Descriptors B.6.0 [Logic Design]: General. C.1.3 **[Other Architecture Styles]:** Data-flow architectures. C.4 **[Performance of Systems]:** Performance attributes. F.2.2 **[Non numerical Algorithms and Problems]:** Sequencing and scheduling. D.1.3 **[Concurrent Programming]:** Parallel programming

General Terms Management, Measurement, Performance, Design.

* *This work was done while Oded was at the Technion.*

Keywords *Scheduling and task partitioning; Shared memory; Parallel processor; Data dependencies*

1. Introduction

We consider programs that comprise a set of serial tasks along with a set of scheduling relations among them. These may represent data dependences and ensure correct execution, or aim to govern the scheduling for other reasons such as efficient resource utilization. This programming model is sometimes referred to as *task-oriented programming*. It is essentially a coarse grain (task granularity) dataflow machine.

The partitioning of a program into tasks aims to expose parallelism and enable the exploitation of many compute cores. When considering two tasks, one of the following holds:

- The two tasks carry out the same operations but on different data (data parallelism). E.g., summing up different rows of a matrix.
- The two tasks perform different operations using the same data (program parallelism). E.g., searching for different virus signatures in the same data.
- The two tasks perform different operations on different data (unrelated tasks).

We consider a shared-memory (no private caches) many-core architecture. A program comprises a set of serial tasks along with a set of precedence relations among them, which represent data dependences and ensure correct execution.

For reasons such as programming convenience and reduced code foot print, multiple-instance ("duplicable") tasks are used in data-parallel situations such as summing up the rows of a matrix. Tasks are dispatched to cores by hardware within very few clock cycles and at a very high rate. This is thus a dataflow machine at the inter-task level, with conventional control flow within each task. The Plurality Hypercore [1, 2] is such an architecture.

XMT (Explicit Multi-Threading) architecture [3, 4] also supports fast dispatching of duplicable tasks to the many-core system. Each of the XMT's cores, however, has a private cache, so cache coherency protocols are needed. The referred XMT is not the same as the Cray XMT which is an entirely different system.

The precedence constraints guarantee correctness, and the absence of private caches obviates the need to consider which core should execute any given task. However, one must still decide the dispatching order whenever the number of runnable tasks exceeds that of available cores. This choice among correct execution orders can impact performance: 1) it can mitigate bottlenecks, namely situations wherein a task that must precede many others is scheduled later than it could have been and now causes cores to be idle awaiting its completion, and 2) it can impact the instantaneous memory footprint of the program and its data, thereby affecting the hit rate of the shared cache.

For a given number of cores and a specific program with known task execution times, one could simply add precedence relations in order to enforce the desired scheduling order. This, however, is more difficult in the general case wherein some of the runtime parameters such as execution time are data dependent. The problem is most acute with duplicable tasks, as the "basic" precedence constraints apply jointly to all task instances.

For synchronization purposes, each duplicable task has an entry point (fork) and an exit point (join), as depicted in Figure 1. The entry point states that all replicas can be dispatched; this can be thought of as a fork. The exit point refers to the fact that the duplicable task is considered complete only after its replicas have been completed; this can be thought of as a join.

Using duplicable tasks has several advantages: task graphs are simple to create, changing the number of replicas is simple (allowing portability), and efficient task dispatching. On the other hand, there are also limiting factors when using duplicable tasks instead of regular tasks and applying scheduling constraints jointly to all the replicas of a duplicable task. These include a loss of expressive power, out of order completion and reduced portability when optimizations have been made.

This work focuses on scheduling constructs ("directives") that can be used by programmers and by automatic optimization tools to further direct the runtime dispatcher, with special attention to duplicable tasks. Such constructs must express relations that occur in real programs and whose translation into scheduling directives impacts performance. Yet, they must lend themselves to efficient implementation in hardware. We present several such directives, along with illustrative examples in which they increase performance. We also outline their implementation in the context of a Hypercore-like system, thereby proving them to be practical.

The remainder of the paper is organized as follows. We briefly discuss related works on scheduling problems. Section II presents Hypercore in some detail. In Section III we consider regular tasks, and present Start-After Start (SAS), a new type of scheduling directive that offers a level of priority that lies between precedence and priority. In Section IV we present our main contribution, new scheduling directives for duplicable tasks. In Section V we present a simple hardware mechanism that supports the new scheduling directives. Finally, Section VI offers concluding remarks.

1.1 Related Works

In this subsection we give a short overview of scheduling and scheduling schemes. We give this discussion for the sake of completeness, yet, we note that our work does not focus on scheduling schemes; rather it focuses on creating scheduling directives that increase the expressive power available to the programmer and application designer.

Scheduling has received much attention in the past half century due to its significance. Two seminal papers, the first by Graham [5] presented scheduling anomalies that can occur when there is a slight change to the program and system parameters and the second by Ullman [6] showed that even some of the simplest scheduling problems are NP -complete.

While this might imply that all scheduling problems are NP-complete, this is not the case. The time complexity of a

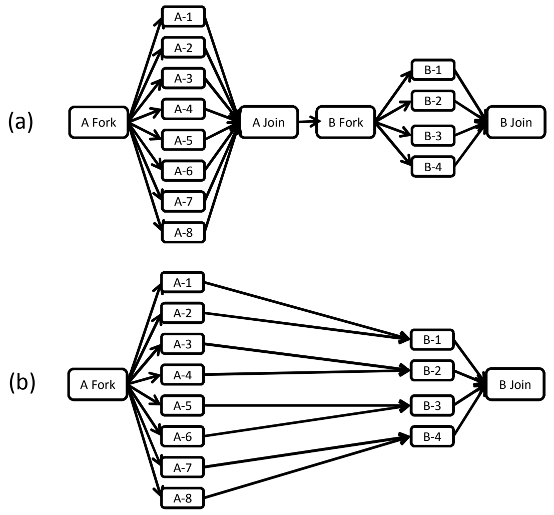


Figure 1 - (a) Two duplicable tasks A, B. A precedes B. A has 8 replicas and B has 4 replicas. (b) Same duplicable tasks with precedence constraints among replicas of the two tasks.

scheduling scheme is based on numerous parameters and constraints which include: the optimality criteria, systems parameters, and the ability to do context switches. These characteristics are reflected by 3-field problem classification $\alpha|\beta|\gamma$ that can be found in Graham et al. [7]. An extended list of scheduling schemes based on this classification can be found in Brucker [8] and Sinnen [9].

It is well worth noting that many scheduling schemes work offline and assume having knowledge on the tasks that make up the applications. As such, they have the ability to make wiser scheduling choices as they are able to go over the full or partial search space. In contrast, online schedulers have to make the scheduling decisions quickly as otherwise the system utilization can be significantly reduced. Furthermore, online schedulers usually have less information on which they base their decision to dispatch a specific task. Because of this, online scheduling schemes require simplicity and make scheduling decisions that are not optimal. Further, hardware schedulers require even a higher level of simplicity.

Applications can be represented using Directed Acyclic Graphs (DAG) in which the vertices represent the tasks and the edges represent the task interdependencies. The incoming edges are known as the tasks dependencies or its precedence constraints. The dependences can be for either control flow or data flow. The vertex weight is the expected amount of time that the task will need for its execution. The edge weights refer to the communication costs between two tasks. By scheduling tasks that have sort of communication cost between them on the same processor, the communication can be avoided as the data has already been fetched.

The critical path of the DAG is defined as the longest path between any entry vertex (vertices that do not have any dependencies) and any exit vertex (vertices that are not dependencies for other tasks). Two notable papers that deal with critical path scheduling problem are by Sih and Lee [10] and Kwok and Ahmad [11]. The first of these, schedules the tasks to a system with a given number of processors whereas the second uses a scheduling scheme that can introduce new processors. As such, the later does not have a limit on the number of processors that it can use for scheduling the tasks.

Gillies and Liu [12] discuss the use of "or" precedence constraint that allows for dispatching tasks only when a partial subset of their precedence constraints have been met. One can consider the case of redundancy where tasks are executed more than once for the sake of correctness. If a task fails then all the dependent tasks cannot be dispatched. For the sake of brevity we don't present additional scheduling schemes that use this type of constraint, yet, we note that this is a scheduling directive.

2. The Hypercore Architecture

Hypercore, developed by Plurality [1, 2], offers a shared-memory many-core system where the on-chip cache is fully shared among the cores, but is partitioned into numerous banks, and a low-latency, high-bandwidth combinational multistage interconnect carries the core-cache traffic. Address interleaving is used for load balancing and collision mitigation. Same-address writes, as well as same-bank accesses (to different addresses within the bank) are serialized by the interconnect. The memory banks are equidistant from all the cores, so this is a UMA system. The absence of private caches (and a large amount of state in them) and the UMA architecture permit any core to execute any compute task with equal efficiency. This greatly simplifies programming and runtime management. The fast task dispatching permits the beneficial exploitation of fine grain parallelism. Typical shared-cache size is several Megabytes.

The programming model is a set of serial tasks along with precedence relations among them. Any task whose precedents have been met is runnable. (It is the programmer's responsibility to provide all the precedence constraints that are required for ensuring correctness.)

Hypercore can be viewed as the "dual" of a single-core processor with out of order execution: the latter is a control flow machine when viewed from afar, and a dataflow machine when one zooms in on the instructions currently in the CPU; Hypercore, in contrast, is a coarse-grain (task granularity) dataflow machine and a fine-grain control-flow machine (each core is typically an in-order pipelined machine). A directed task graph is used to express the desired precedence relationships among tasks.

Plurality has implemented an online hardware scheduler called the "Synchronizer\Scheduler". It receives the task graph along with pointers to the start address of every task, tracks the completion of every task, and dispatches a runnable task as soon as a core becomes available. (We will simply refer to this unit as the dispatcher or the scheduler.)

To enable fast scheduling and dispatching, Plurality created a distribution network between the dispatcher and the cores. From the moment the dispatcher dispatches a task until the task reaches an idle core, it takes $O(log(cores))$ cycles. The dispatch network is a tree rooted at the dispatcher with the cores as the leaves. The dispatcher node's fanout is implementation dependent. Each internal node in the dispatch network can forward the dispatch to one of its childrens in one cycle.

The dispatcher, implemented in hardware, is very fast (in terms of both latency and throughput). The fast dispatcher enables beneficial exploitation of fine-grain parallelism. Together with the UMA, high-bandwidth shared-cache, this yields a very effective, agile and easy to program architecture.

Each of the nodes in the scheduler's distribution network can complete its mission in one cycle, sending the dispatch request onward. Hypercore supports two types of dispatching: 1) dispatching a single task on each sub-tree in the distribution network; this limits the number of dispatched tasks per cycle to

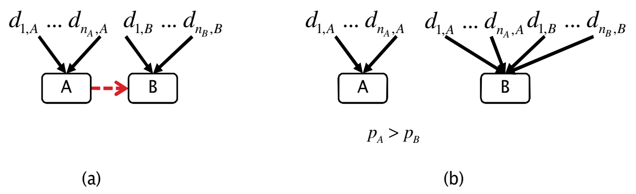

$d_{1,A}$ ... $d_{n_A,A}$ $d_{1,B}$ ... $d_{n_B,B}$
A
B
$d_{1,A}$ ... $d_{n_A,A}$ $d_{1,A}$ ... $d_{n_A,A}$ $d_{1,B}$ ... $d_{n_B,B}$
A
B
$p_A > p_B$
(a)
(b)

Figure 2 - (a) Desired graph. The dashed red arrow represents the SAS requirement between tasks A and B. (b) SAS implementation, adding A's precedences to B.

the number of sub-trees; 2) dispatching a duplicable task with multiple copies on each sub-tree; here, the number of dispatched replicas is limited only by the total number of cores in that sub-tree.

It is Plurality's goal to make this system a low power system. While exact numbers cannot be given as this platform has not been fully synthesized at the date of submission, the numbers suggest $\sim 4\ Watts$ for 64 OpenSPARC cores at $500MHz$ with $40nm$ CMOS technology.

Hypercore is extremely attractive from a power-performance, applicability and ease of programming perspectives. We therefore chose it as the basic architecture for our work, though much of our findings and suggestions have broader applicability.

3. Scheduling Directives for Regular Tasks

We next present scheduling directives. For convenience, we express them in the context of task B that depends in some sense on a set A of "prerequisite" tasks. Also, we use p_T to denote the priority of task T; a larger number represents higher priority.

3.1 Conventional directives (not new)

Start After Complete (SAC). This is simply the precedence relation, whereby B may be dispatched only after all tasks in A have been completed. We used $Prec(B)$ to denote the set of tasks on which task B has a SAC dependence.

Priority. Here, if both B and some task in A are runnable but there is only one available core, A will be dispatched. If, however, only B is runnable, it need not wait for A to be dispatched.

We next present a new scheduling directive for regular tasks.

3.2 Start After Start (SAS)

The SAS(B,A) directive

Here, task B may not be dispatched until all A tasks have been dispatched. SAS expresses an intermediate degree of prioritization, with SAC being more strict and Priority being less strict. Note that SAS is not "work conserving," as a core may be kept idle despite the fact that there is a runnable task.

A motivating example for this is a situation wherein B is runnable and A is not. Also, certain other tasks have a SAC dependence on A, whereas none depend on B. Suppose that a single core becomes available, and A becomes runnable shortly thereafter. Although B does not depend on A, letting B grab the core may delay the dispatching of A if A becomes runnable but there is no core to run it on. In this case, it may be desirable to require that B be dispatched only after A has been dispatched, hence the term "Start after Start".

SAS Implementation

SAS(B,A) can be expressed using SAC and Priority as follows:

- Set $Prec(B) \leftarrow Prec(B) \cup Prec(A)$

- Set $p_A > p_b$

The first step ensures that B does not become runnable before A, and the second step ensures that if both are runnable then A is dispatched first. Clearly, both elements are necessary. SAS implementation can thus be based on the existing mechanisms for SAC and priority. Figure 2 depicts this.

We next consider the case of duplicable tasks.

4. Scheduling Directives for Duplicable Tasks

4.1 Need and Challenges

SAC at the entire-task granularity suffices for guaranteeing correctness. However, it may be overly restrictive and limit parallelism, as illustrated by the following example. Unless stated otherwise, we use a letter to refer to a duplicable task and a subscript value for each of its replicas.

Example 1. Suppose that B_i only depends on data computed by A_i and by A_{i-1}. Clearly, there is no need to wait with the dispatching of B_1 until all replicas of A have been completed. It is therefore desirable to express the SAC constraints between duplicable tasks more precisely in order to expose more parallelism.

Even in the absence of data dependence among duplicable tasks, it may be important to coordinate the dispatching of their replicas. One possible reason is memory performance, as illustrated by the following example.

Example 2. Consider A and B, duplicable tasks that both access the same elements of a data array X in the same order. X is larger than the shared cache. If all of A's replicas are executed prior to any of $B's$ replicas, every data element would have to be brought into the cache twice, as it would drop out of the cache prior to its use by B. If, instead replicas of A and B were executed concurrently (in lockstep or nearly so), this situation could be avoided.

Remark. One may wonder why the work should be partitioned into two tasks in the first place. The answer is that so doing exposes more parallelism. This is important both when the number of cores exceeds the number of data elements, but also serves to reduce the required cache size in support of any given level of computational parallelism. Another use case is when the tasks cannot be fused together as discussed in Section 4.2 .

In view of the above, it is clearly desirable to be able to coordinate the execution of replicas of duplicable tasks with finer granularity. Also, it may be desirable to "throttle" the dispatching of replicas of a single duplicable task for reasons such as total instantaneous memory footprint.

One could conclude from the above that perfect lockstep is the solution, at least SAS lockstep (i.e., controlling the dispatch order). However, this is somewhat simplistic. In Hypercore, for example, the rate at which same-task replicas can be dispatched is much higher than the dispatch rate of different tasks. It is therefore desirable to permit bursts of same-task replica dispatching, but to control burst length.

The challenge is to try and provide sufficient expressive power for stating the desired inter-replica constraints and pacing while permitting sufficiently simple implementation in terms of both speed and the amount of dynamic state information that must be kept. We next present such extensions of SAC and SAS, as well as a limit on the number of active replicas. Prior to so doing, we present the required priority and state information to which we have elected to restrict our proposed directives. This is a sensible yet subjective, self-imposed complexity constraint.

Before proposing our scheduling directives, we next consider and assess several approaches.

4.2 Duplicable Tasks Limitations and Workarounds

In this subsection we present several workaround to the inherently limited expressive power available when duplicable tasks are treated as a single entity.

One way of increasing the expressive power for scheduling constraints among replicas of duplicable tasks is to treat each replica as an independent task. E.g., SAC(B_j, A_i). This, however, has major drawbacks: 1) bloated task graph, often becoming impractical for efficient hardware implementation, 2) the graph is parameter dependent: a change in the number of replicas (a parameter change for the application) requires a change to the graph; 3) Dispatching regular tasks is usually less efficient than the dispatching of duplicable tasks, because the scheduler [2] can dispatch several replicas of a duplicable task in a single cycle vs. a single regular task per cycle. This approach is therefore impractical.

Another possible approach entails fusing two duplicable tasks A, B into one new duplicable task C. As the task graph designer knows the relationships between the tasks, the task graph designer may be able to redesign the graph accordingly. However, task fusion suffers from several deficiencies: 1) the duplicable tasks may not have an equal number of replicas, making the fusion more complicated as in Fig. 1. 2) Even simple relationships between the replicas of the two duplicable tasks A and B can be hard to fuse into a new duplicable task C. For example, given that B_i is dependent on A_i and A_{i+2} , which replica of C should compute A_j? Should C_j or C_{j+2} compute it? Both C_j and C_{j+2} can compute A_i and A_{i+2}. This causes redundancy in operations. For the scenario that B_i depends on a large number of tasks, this approach is intolerable. Another solution is to let C_j compute A_{j+2}. Due do to out-of-order completion, C_{j+2} now becomes dependent on C_j, and therefore it cannot be dispatched until the completion of C_j. While task fusion may be suitable for some problems, specifically when B_i is dependent only on one A_i , it is not suitable for many scenarios.

We next present our proposed approach and specific scheduling directives. We begin by stating the required state in formation and presenting the required taxonomy.

4.3 Priority and State Information

Priority

A replica of a duplicable task inherits the priority of its task. Additionally, it has an intra-task priority (relative to other same-task replicas), which is normally highest for the lowest-number replica. We furthermore assume in-order dispatching of same-task replicas. Specifically, in-order submission to the dispatch dissemination tree-like interconnect. (The exact order in which they obtain cores is not critical for correctness.)

State information

For each duplicable task, A, we keep the following state information, which are updated by the online scheduler:

- $A.n$ – total number of replicas (static). Supported by Hypercore.
- $A.s$ – number of dispatched/started replicas (both active and completed). Supported by Hypercore.
- $A.c$ – number of completed replicas. Supported by Hypercore.

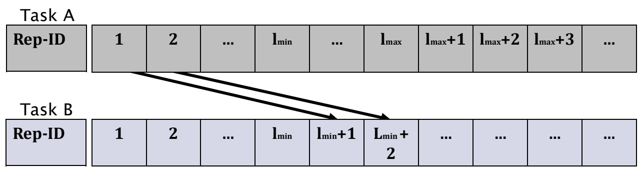


Figure 2 - SAC Type 1 for duplicable tasks. Edges from A to B are precedence.

- $A.es$ – the "earliest" (lowest index) replica that has been started (dispatched) but has yet to be completed. In Section 5. we show how to implement its computation efficiently in a manner that is similar to a re-order buffer.

Lemma 1: Consider a duplicable task A. If $A.es \geq A.n$, then all replicas preceding A_n have completed.

Proof: the in-order dispatching of same-task replicas ensures that these replicas had all been dispatched, and the fact that *es* is the index of the earliest replica that has not been completed ensures that there are no active lower-index replicas. □

The number of active replicas of a given task A is $A.s$-$A.c$. The indication for entire duplicable task completion is $A.c$=$A.n$.

In the upcoming subsections, the following functions will be used on replicas of A to determine its status:

- $\mathrm{S}(\mathrm{A_i})$ – returns true iff $\mathrm{A_i}$ has started,
- $\mathrm{C}(\mathrm{A_i})$ – returns true iff $\mathrm{A_i}$ has completed,
- $\mathrm{D}(\mathrm{A_i})$ – returns true iff $\mathrm{A_i}$ may be dispatched.

In addition to the per-task state, two parameters, l_{min} and l_{max}, are optionally used to constrain the permissible progress "gap" between any two duplicable tasks. Different directives will use variables with different exact meanings, so more precise definitions will be given in context.

4.4 Start After Complete (SAC) for duplicable tasks

Applying directives to entire tasks is the simplest to implement but hides legitimate parallelism, as it often creates false dependences among the replicas of the different tasks. At the other extreme, expressing the exact inter-replica dependences exposes all available parallelism but is complicated to express and track.

In this section, we provide extensions of SAC to duplicable tasks. They represent a trade-off between complexity and expressive power. They are moreover intended mainly for situations in which the set of task A replicas on which $\mathrm{B_{i+1}}$ depends is obtained from the set on which $\mathrm{B_i}$ depends by adding one to the index of every replica in the latter set. Also, we assume in-order dispatching of same-task replicas, though out-of-order completion is permitted.

Consider duplicable tasks $\mathrm{A, B}$ such that $\mathrm{B_i}$ depends (SAC) on some set of $\{\mathrm{A_j}, \mathrm{A_k} \ldots \mathrm{A_{max}}\}$ of A's replicas, where $\mathrm{A_{max}}$ is the replica with the highest id on which $\mathrm{B_i}$ depends. ($\mathrm{B_{i+1}}$ depends the set of $\{\mathrm{A_{j+1}}, \mathrm{A_{k+1}} \ldots \mathrm{A_{max+1}}\}$ and so forth). Our SAC directive simplifies this by requiring that all replicas of A up to and including $\mathrm{A_{max}}$ be completed as a prerequisite for the dispatching of $\mathrm{B_i}$. The price is "false" constraints, but correctness is maintained. Moreover, we will later show a potential performance advantage, namely the ability to dispatch bursts of same-task replicas. In Hypercore, this is much more efficient that dispatching individual replicas or ones belonging to different tasks.

We next develop the underpinnings of a simple expression and implementation of this constraint.

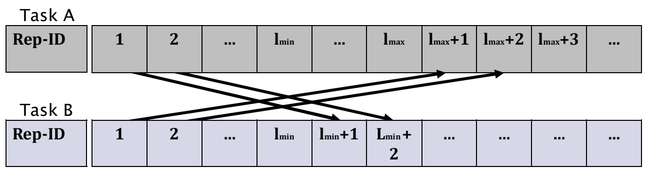


Figure 3 - SAC Type 2 for duplicable tasks. Edges are precedence constraints. Note that the precedence constraints are in both directions

Theorem 2: If A.$es \geq max$ such that A_{max} has completed, then $\mathrm{B_i}$ may be dispatched.

Proof: by Lemma 1, $es \geq max$ ensures that A's replicas with a smaller index than *max* have completed, and together with the fact that A_{max} has also completed this guarantees that all the prerequisites for the dispatching of B_i have been met. □

Corollary 3: Correctness can be guaranteed by way of a single SAC constraint, namely $B_i \leftarrow A_{max}$ along with an indicator for the additional constraint on the value of *es* (whose value equals *max*). Moreover, whenever the precedence constraints are relative (a fixed function of i) and satisfaction of the constraints for B_i implies that they have been satisfied for all earlier replicas of B, the constraints can be updated on the fly for different values of i. □

Remark: due to the possibility of out-of-order completion of same-task replicas, *es* may increase in arbitrary increments. A reorder-buffer technique can be used for handling updates to *es*. We will return to this in Section 5.

The SAC(B,A, *l*) directive

Herel, l is the difference between the index of a replica of B and that of the highest-index replica of A on which it has a SAC dependence. In other words, this is a "relative" or "sliding window" equivalent of the situation presented at the beginning of this subsection. Figure 4 depicts an example.

The directive is stated formally in (4.1) and (4.2). The former ensures in-order dispatching of A's replicas; the latter ensures in-order dispatching of task-B replicas and expresses the actual constraint. Dependences on negative-index replicas are taken as having been met.

$$D(A_i) \leftarrow \{S(A_{i-1})\}; \tag{4.1}$$

$$D(B_j) \leftarrow \{S(B_{j-1}) \wedge C(A_{j-l})\}. \tag{4.2}$$

Creating hardware to compute these rules is simple as A's replicas are dispatched like any duplicable task. We denote $dispatch_B$ as the number of $B's$ replicas that can be dispatched at any given time. $dispatch_B$ is computed based on (4.2) :

$$dispatch_B = A.es - B.s - l \ . \tag{4.3}$$

The first part of the expression $A.es - B.s$ refers to the distance between the earliest active in A and the last replica to start in B. This distance has to be at the very least l for there to be replicas of B that can be dispatched.

Remark. This directive can use the distribution network efficiently to dispatch multiple replicas concurrently, as the value of $dispatch_B$ can be greater than one.

Multiple SAC constraints

It may often be the case that replicas of one task depend on those of several other tasks. In fact, there may even be

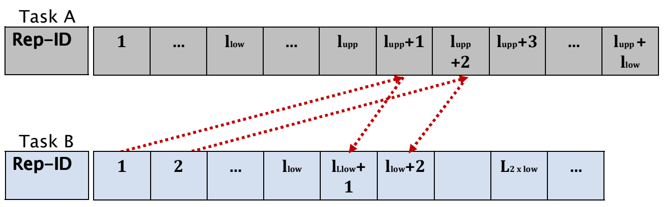


Figure 5 - SAS for duplicable tasks. The SAS (dashed red) edges are in both directions.

bidirectional dependence among replicas of two given tasks. Figure 3 depicts an example.

For any given task, its constraints are simply the union of all the SAC constraints that express its dependence on other tasks (or even on its own earlier replicas, for that matter). The number of dispatchable replicas of such a task is the minimum over the numbers computed based on the individual constraints.

Deadlock

Deadlock can and should be checked for statically (no need for dynamic checking) using the established techniques (looking for loops in the replica-granularity SAC-dependency graph). Special caution must be taken only whenever the limited expressive power of our SAC constraints result in the implicit addition of (false) constraints, which may cause deadlock in situations that were originally fine.

4.5 Start After Start (SAS) for duplicable tasks

Given two duplicable tasks, A and B, possibly with no data dependence between their replicas, SAS(B,A) is aimed at specifying the level of synchronization ("lockstep") between the two tasks. Specifically, it specifies the permissible range of the number of replicas by which A's dispatching may advance over B's dispatching. Using the parameters l_{min} and l_{max}, the definition of this directive is as follows.

Definition: $SAS(B, A, l_{min}, l_{max})$

1. The next replica of B may be dispatched only if $A.s - B.s > l_{min}$,
2. The next replica of A may be dispatched only if $A.s - B.s < l_{max}$,
3. $l_{\max} \geq l_{min} \geq 0$. Negative numbers can be thought of as switching the roles of A and B.

We refer to (l_{min}, l_{max}) as the *range*.

As stated at the beginning of this section, the purpose of SAS is not correctness. Rather, it is aimed at improving resource utilization and efficiency of operation. In addition to the aforementioned memory-access advantages, SAS can be used to increase the likelihood of being able to take advantage of the burst dispatching capability of Hypercore for same-task replicas. This will be mentioned later in some more detail.

Example 3: Perfect lockstep with priority to replicas of A: range=(0,1) and set $p_{A_i} > p_{B_{i+1}} \wedge p_{A_{i+1}} < p_{B_{i+1}}$. Note that the priorities alternate between the duplicable tasks and that the indices used in this example are intended for this example alone. (In this case the tasks themselves would receive identical priorities.)

Note that dispatching task B replicas depends on the dispatching of A′s replicas and vice versa. Without the former, it would be possible to dispatch all the replicas of B without dispatching a single replica of A, and vice versa.

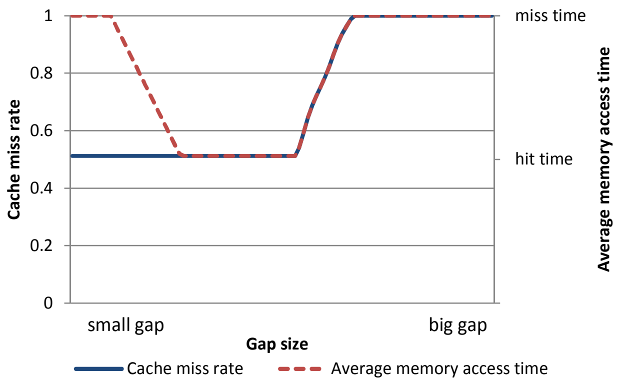


Figure 6 – Synthetic plots that illustrates the effects of gap size on cache performance.

Perfect lockstep is not recommended, as it often prevents the dispatching of bursts of same-task replicas, an operation that has much higher throughput than the dispatching of individual tasks or replicas.

The SAS constraints are stated formally below. Figure 5 depicts the SAS directive. (4.4) enforces the in-order dispatching of replicas of A, and prevents A from getting ahead of B by more than l_{max} replicas (in terms of dispatching, not completion). Similarly, (4.5) ensures in-order dispatching of replicas of B, and prevents B from trailing A by fewer than l_{min} replicas. In both cases, a dependence on a negative-index replica is taken as satisfied.

The constraints on the replicas are formalized as following.

$$D(A_i) \leftarrow \{S(A_{i-1}) \wedge S(B_{i-l_{\max}})\}. \tag{4.4}$$

$$D(B_j) \leftarrow \{S(B_{j-1}) \wedge S(A_{j+\min})\}. \tag{4.5}$$

In (4.6) and (4.7) we show that dispatching replicas meeting the constraints of (4.4) and (4.5) can be done efficiently. The following two expressions compute the number of replicas from each of the duplicable tasks that can be dispatched:

$$dispatch_A = l_{\max} - (A.s - B.s) \tag{4.6}$$

$$dispatch_B = (A.s - B.s) - l_{\min}. \tag{4.7}$$

Note that when the gap is inside its permissible range, replicas of either task may be dispatched. Here, by assigning higher priority to one of the tasks, burst dispatching will be used whenever possible. Specifically, if the number of available cores is smaller than the difference between the current gap and the relevant limit, a single burst of same-task replicas will take place, which is the most efficient.

Remark. It is possible to design the scheduler such that, when replicas of multiple tasks are dispatchable, it would favor tasks based on the number of dispatchable replicas, the number of available cores, and the distance to the relevant limit on the gap. Details are beyond the scope of this paper.

Example 4: In this example we show a use case for SAS that allows for utilizing the scheduler's efficient burst scheduling. For simplicity, we assume a system with two cores. There are two duplicable tasks, A and B, such that all their replicas execute in unit time. Assume that $|A| = |B| \gg 2$. We set $l_{min} = 2$ and $l_{max} = 4$. At time $t = 0$, we dispatch A_1 and A_2. At time $t = 1$, we check if replicas of B can be dispatched: $dispatch_B = (2 - 0) - (2) = 0$ and $dispatch_A = 4 - (2 - 0) = 2$. Therefore, at time $t = 1$ we dispatch A_3 and A_4. At time $t = 2$: $dispatch_B = (4 - 0) - (2) = 2$ and $dispatch_A = 4 - (4 - 0) = 2$. Therefore, at time $t = 2$ we dispatch B_1 and B_2. At time $t = 3$: $dispatch_B = (4 - 2) - (2) = 0$ and $dispatch_A = 4 - (4 - 2) = 2$. Therefore, at time $t = 3$ we dispatch A_5 and A_6. This will repeat itself until all the replicas have been dispatched.

The above example is somewhat artificial, as the probability of cycle-accurate lock-step is very low. However, the situation wherein it suddenly becomes possible to dispatch several tasks is quite plausible. It arises, for example, when implementing a barrier. Until the barrier condition is met, cores may become idle as no post-barrier tasks may be dispatched. Once the final remaining barrier condition is satisfied, those cores become usable simultaneously. Using our terms, if both task A replicas and task B replicas have a SAC dependence on task C, a high-priority task that has a SAC dependence between C_i and C_{i-1} and is thus executed sequentially, the completion of a replica of C would possibly render multiple replicas of A and B dispatchable simultaneously to a set of idle cores.

In summary, SAS is a useful construct for pacing the relative progress of different duplicable tasks, in support of fair resource allocation, reduced memory footprint, and more effective dispatching.

4.6 More on SAS and Memory Performance

The obvious benefit of using SAS to coordinate the progress of two duplicable tasks whose replicas operate on the same data is the reduction of the instantaneous memory footprint and a reduction in the miss rate of the shared cache. Indeed, letting the progress gap between the active duplicable tasks grow eventually results in a situation whereby data brought into cache by task A is removed from the cache before task B uses it. The result is that both tasks, rather than only A, incur a cache miss.

A simplistic implication of the above observation is that the best memory performance is attained when tasks operating on the same data are paced in perfect lockstep. However, this is not the case.

One reason for not forcing perfect lockstep, mentioned earlier, is dispatching efficiency. However, in certain cases there is also a memory related reason.

The problem is that if task B wishes to access a cache line that was just requested by task A, and the latter incurred a cache miss (possibly a compulsory one), B would not incur a miss; however, it would still have to wait for the data to arrive. Therefore, the memory access time experience by B would be very similar to the miss time. If, instead, B were delayed some, the data would be in the cache.

This phenomenon is illustrated schematically in Figure 6. The abscissa is the gap size, and the two ordinates are cache miss rate and average memory access time. We see that whereas the miss rate is monotonically non-decreasing with gap size, average memory access time has a sweet range. The figure is for illustration purposes, not representing actual results, and is intentionally not calibrated.

The following is suggested as a rule of thumb for selecting l_{min} for the case that the memory needed by each replica is considerably smaller than the shared memory and the replicas of both duplicable tasks have the same execution times:

$$\tilde{l}_{\min} = 2 \cdot |Cores|. \tag{4.8}$$

In view of the above, there are several good reasons for imposing both an upper limit and a lower limit on the progress gap between two tasks that operate on the same data.

4.7 A Basic Simulation Study

In this section, we present a simple performance evaluation. As Plurality has yet to ship the Hypercore system and we had limited access to the actual hardware implementation, we created a simulator to test SAS and the additional directives. We used a queue-based simulator that dispatched duplicable tasks to a shared-memory many-core system similar to the Hypercore. Using Plurality's cycle accurate simulator, we were able to obtain cycle counts for memory accesses in the cache, and for the execution of both floating point and integer operations. Plurality's cycle accurate simulator did not take into account the DRAM memory. For DRAM access time, we used actual DRAM times of current technologies. These numbers were passed on to our simulator. Our simulator used the following parameters: 64 cores with a $2MB$ shared cache. We used 32 byte cache lines and a direct-mapped cache. The latency for fetching data out of the DRAM was 20 cycles and fetching from the cache was 2 cycles. Should the 20 cycles latency be an under estimation, this would result in the performance being even more sensitive to cache misses.

The application that we tested was the computation of x and y derivatives of a 2000×2000 matrix of single byte elements, which is typical of gray-scale image processing. The size of the array is 4MB, so the array cannot fit into the shared on-chip cache. The first duplicable task computed the x derivative, and the second duplicable task computed the y derivative for the same matrix. The duplicable tasks were implemented at a fine granularity – element level. Thus, each duplicable task had $n = 4 \cdot 10^6$ replicas, which is the number of elements in the array.

In Figure 7 and Figure 8, we present two plots of the number of cache misses and the runtimes for an application, respectively. The solid curve corresponds to the two duplicable tasks not running concurrently. The dashed curve corresponds to executing the tasks concurrently, governed by the SAS directive. The abscissa is the SAS-imposed gap size. The ordinate in Figure 7 is the total number of caches misses, and in Figure 8 it is the number of cycles required to complete the application.

It can readily be observed that the use of SAS with a sufficiently small gap offers a noticeable improvement relative to the serialization of the two tasks. This is due to the reduction in cache miss rate. When the gap is large, there is no performance improvement, because data brought into the cache by A is evicted before B has a chance to use it, so B also incurs misses.

4.8 Additional directives

We have presented and discussed two scheduling directives for duplicable tasks: SAS and SAC. We next briefly present several additional structured scheduling directives that we believe to be useful for task graph designers:

- *Limit Number of Active Replicas (LNAR)* is used in order to limit the number of concurrent replicas of a duplicable task to K. This is useful whenever the number of replicas exceeds the total number of cores, and it is desirable that not all the cores execute replicas because of I/O limitations or memory footprint issues.

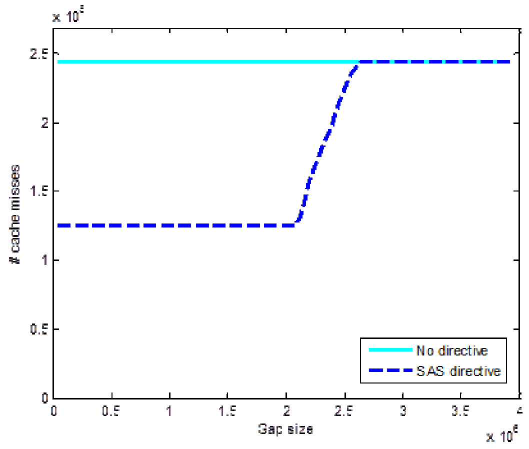


Figure 7 – Total number of caches misses vs. the SAS imposed gap size for two duplicable tasks with a similar (not exact) access pattern (dashed). The case of no SAS (solid), for which there is no notion of a gap, is brought as a baseline for comparison (solid line).

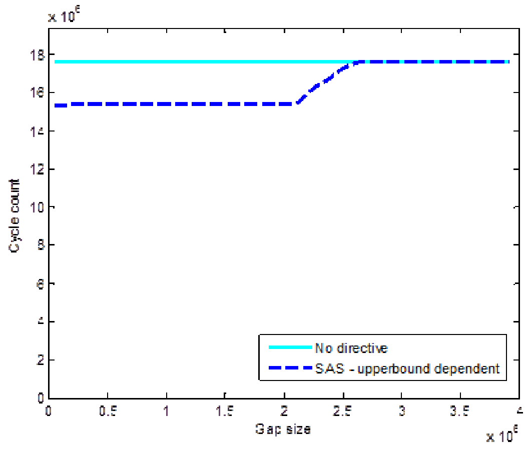


Figure 8 - Execution time vs. SAS gap size for two duplicable tasks with a similar (not exact) access pattern (dashed), The case of no SAS constraints (solid) is the baseline..

The first K can be dispatched without any constraint. For all $i > K$ the following constraints are added.

$$S(A_i) \leftarrow \{S(A_{i-1}) \wedge (A.s - A.c < K)\} \tag{4.9}$$

The number of replicas that can be dispatched at a given time is:

$$dispatch_A = K - (A.s - A.c) \tag{4.10}$$

- *Assign Cores Fairly (ACF)* is used in order to split the cores evenly between two duplicable tasks A and B. This directive is useful when the number of replicas exceeds the total number of cores and it is desirable that not all the cores execute same-task replicas. This directive refers only to the number of started replicas and not to the order of their completion. The constraints on A are:

$$S(A_i) \leftarrow \left\{ \begin{matrix} S(A_{i-1}) \wedge \\ (A.s - A.c) \le (B.s - B.c) \end{matrix} \right\} \tag{4.11}$$

The first constraint on A is the usual in-order dispatching of same-task replicas. The second constraint ensures that A has fewer active tasks than B does, Due to symmetry, the constraints on B are the same.

To compute the number of replicas that can be dispatched:

$$dispatch_{A/B} = (A.s - A.c) - (B.s - B.c). \tag{4.12}$$

In expression (4.12) $t$he numbers of active of replicas of both tasks are compared. If $dispatch_{A/B} < 0$ then there are more active replicas of B in the system and A may dispatch accordingly the difference. If $dispatch_{A/B} > 0$ then there are more active replicas of A in the system and B may dispatch accordingly the difference. If $dispatch_{A/B} = 0$ then there is an equal number of active replicas and the idle cores should be divided equally between the duplicable tasks.

- *Limit Number of Replicas after Earliest Started (LNR)* is used in order to limit the span (range of ids) of active replicas of a given duplicable task. This can be seen as a limited size sliding window of dispatched replicas. Until the first replica in the window, es is completed, the window cannot be moved forward. This directive is similar to LNAR with the difference being that LNR limits the number of dispatched replicas w.r.t. to the es replica dispatched. Furthermore, this directive enforces correctness unlike LNAR.

The first K replicas can always be dispatched. For all $i > K$ the following constraints are added.

$$S(A_i) \leftarrow \{S(A_{i-1}), C(A_{i-K})\} \tag{4.13}$$

The number of task-A replicas that can be dispatched at a given time is:

$$dispatch_A = K - (A.s - A.ea) \tag{4.14}$$

- Start After Merged Completion (SAMC) is used to state that the prerequisites between the duplicable tasks are such that each B_i is dependent on the completion of M consecutive task-A replicas. Different task-B replicas are dependent on disjoint subsets of task-A replicas. Given two duplicable tasks, A and B, the dependency between the replicas can be defined as $B_j \leftarrow A_{M\cdot j}, A_{M\cdot j+1}, \dots, A_{M\cdot(j+1)-1}$. This directive would be useful in implementing a task graph for merge-sorting [13].

Task-A replicas are unconstrained except for in-order dispatching. The constraints on task-B replicas are as follows:

$$S(B_j) \leftarrow \left\{ \begin{matrix} S(B_{j-1}) \wedge C(A_{M\cdot j}) \wedge \\ C(A_{M\cdot j+1}) \dots \wedge C(A_{M\cdot(j+1)-1}) \end{matrix} \right\} \tag{4.15}$$

The number of B's that can be dispatched is

$$dispatch_B = \frac{A.ea}{M} - B.S\ . \tag{4.16}$$

Computing this expression is more demanding than computing the other expressions due to the division operator.

5. Thread Re-Order Buffer

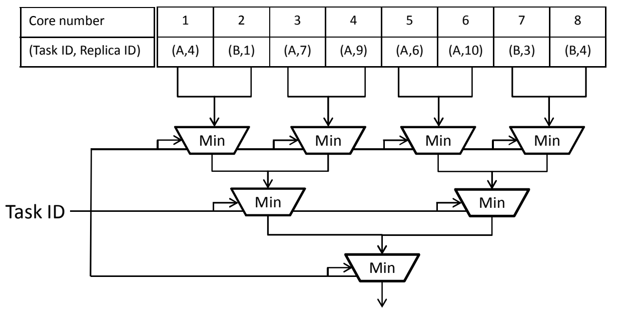
In representing (and enforcing) a SAC constraint between replicas of different duplicable tasks, there are two extremes: 1) a SAC constraint between the entire tasks (all replicas of A must complete before any replica of B is dispatched), and 2) specify the exact inter-replica dependences and enforce them. The former was claimed to potentially reduce performance by hiding too much of the permissible parallelism, and the latter is complex to implement as much state must be maintained and updated. Instead, we proposed a compromise: If a given replica of task B depends on a set of replicas of task A, treat it as if it depends on the completion of all replicas of A with indices less than or equal to that of the highest-index replica of A on which it actually depends. We then presented the state variable (for each task) *es*, which is the index of the lowest-index replica that has been dispatched but not yet completed. We also showed how *es* can be used in conjunction with in-order dispatching of same-task replicas in order to determine whether any given task-B replica may be dispatched. Finally, we pointed out that, due to out-of-order completion of same-task replicas, the value of *es* may change in arbitrary (positive) increments. In this section, we present a scheme for updating the value of *es*. We refer to the value of *es* of a given task A as *A.es*.

5.1 Replica Re-Order Buffer for Updating *es*

Consider a change of $A.es$ from $A.es_{old}$ to $A.es_{new}$. This can only be brought about by the completion event of A's replica number $A.es_{old}$, with replica number $A.es_{new}$ having been dispatched prior to this event and with all replicas in the range ($A.es_{old}+1$, $A.es_{new}-1$) having completed prior to it as well. It is readily evident that the update mechanism of *es* is essentially the same as that for controlling the commit phase in processors with out-of-order execution and in-order commit. Specifically, we can employ a reorder buffer (ROB) [14, 15] per active duplicable task.

For the implementation of es field to be considered efficient and practical, it must meet the $O(\log_2(|cores|))$ dispatch time of the current scheduler and be low power, small in physical size and scalable.

While the function of our task ROB is the same as one of the functions of an instruction ROB, there are some important differences. For example, we don't need to actually do anything with completed replicas, other than move a pointer. Therefore, the maximum number of replicas (jointly for all active tasks) equals the number of cores (or, if the cores can handle a few task concurrently, like multi-threading, a small multiple thereof).



Core number	1	2	3	4	5	6	7	8
(Task ID, Replica ID)	(A,4)	(B,1)	(A,7)	(A,9)	(A,6)	(A,10)	(B,3)	(B,4)

Figure 9 - Schematic diagram of the new hardware. For simplicity the non-pipelined version is presented. Each core maintains the duplicable task ID and the replica ID of the task that it is executing.

TABLE 1
RE-ORDER BUFFER SPECIFICATIONS

Parameter name	Value
Number of cores	64
Process used	65 nm
Total dynamic power	7.0 mW
Physical size	0.025 mm^2
System frequency	$400MHz$
Number of cycle	3

We now present a low-power logarithmic time method for computing the es replica, initially considering a single active task. As depicted in Fig. 9, we create a tree whose leaves are the cores. Each core provides the index of the replica on which it is working, and a null value if it is idle. Intermediate nodes compute the minimum over the numbers (on each) that they receive from their sons and pass it on to their father. The root thus holds the minimum index value of an active replica, which is exactly *es*. This can be constructed as simple combination logic, or can be pipelined. Fig. 9 depicts the simplest case, namely a binary tree without pipelining.

The extension to multiple concurrently active tasks is as follows. Now, each core provides both the task ID and the replica index. In one possible embodiment, depicted in Fig. 10, the tree is replicated several times (equal to the maximum supported number of concurrently active different tasks, and the replica ID's are directed to the relevant tree based on their task ID. Alternatively, a single, time-multiplexed tree can be used. In each clock cycle, the cores (or some logic between them and the first layer of internal tree nodes) receives a task id, and only lets the replica ID through if it belongs to the appropriate task.

Since the number of concurrently active duplicable task is usually small, and since we are dealing with task granularity, the slight additional delay of the time-multiplexed approach is likely to be tolerable. Finally, one can combine the two schemes so as to support a certain number of concurrent tasks with no penalty while not limiting the number of concurrent tasks that can be supported. In Table 1, the specifications of our "virtual" implementation of the hardware is given for a 64-core system. The implementation is low power, low latency and requires little chip space. Further implementation details are left to implementers.

5.2 ROB-Scheduler Interaction

Figure 10 depicts a schematic diagram of how the new thread re-order buffer interacts with the scheduler and the cores. Given the value of es for each task, along with other information available to the Hypercore scheduler (e.g., the next replica to be dispatched for each task as well as the various constraints), the extension of the scheduling logic to make use of es and the constraints as described earlier is a simple engineering task using simple logic. Details are therefore omitted.

We note in passing that the new thread re-order buffer has a similar structure to the scheduler's distribution network, but operates in the reverse direction. In practice, it may be beneficial to co-design the two networks.

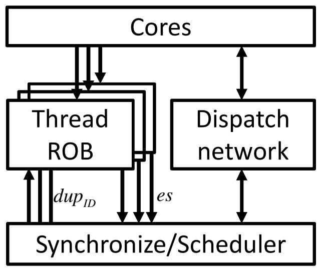


Figure 10 - Block diagram of a the system using a multiple Thread Re-Order Buffers.

6. Conclusions

This work addressed a computing framework whereby scheduling policies are decided offline, and are enforced during run time. Specifically, we investigated the case of duplicable tasks, i.e., tasks that have many instances (data parallelism).

This paper did not offer scheduling policies; instead, it offered directives that serve policy makers (human and tools alike) to express their policies.

Some of the proposed directives serve to express correctness constraints, while others facilitate performance enhancement (e.g., effective use of the memory system) by controlling the relative progress of different duplicable tasks.

The proposed directives represent what we view as a sensible trade-off between expressive power (and resulting benefits) and implementation complexity. To this end, we sketched a power efficient implementation of the main directives.

In our work, we used Plurality's shared-memory many-core system as a reference system for the incorporation of new scheduling directives. The new scheduling directives are not only intended for Plurality's system but can be used for other systems as well. These directives might be useful for NVIDIA's GPU platform with CUDA [16]. More specifically, these scheduling directives would be useful for parallel prefix sum [17] using CUDA. These directives might also be useful for software packages such as the Intel Concurrent Collections (CNC) platform [18].

We presented a small simulation study of a particular application, wherein the performance gain due to the decreasing in cache misses (a 50% reduction in miss rate) is around 15%. We encourage others to find applications of interest that can use these directives.

Our focus was on two directives: Start After Complete (SAC) and Start-After-Start (SAS). However, we presented several additional directives and discussed them briefly.

Finally, we briefly discussed implementation. We showed a ROB-like hardware scheme for updating the "earliest started" (*es*) value of a task, and showed that this and the various constraints can be integrated into an actual system, Plurality's Hypercore system, while maintaining the low power and space envelope using simple logic design. Lastly, we showed that the hardware used for computing the *es* field is conceptually similar to a ROB.

References

[1] Analysis: 'Hypercore' Touts 256 CPUs Per Chip. *EE Times* www.eetimes.com/design/signal-processing-dsp/4017491/Analysis--Hypercore-touts-256-CPUs-per-chip.2007).

[2] HyperCore Software Developer's Handbook *ed: Plurality, Online: www.plurality.com*2009).

[3] Wen, X. and Vishkin, U. Fpga-based prototype of a pram-on-chip processor. In *Proceedings of the Proceedings of the 5th conference on Computing frontiers* (Ischia, Italy, 2008). ACM

[4] Wen, X. *HARDWARE DESIGN, PROTOTYPING AND STUDIES OF THE EXPLICIT MULTI-THREADING (XMT) PARADIGM*. University of Maryland, 2008.

[5] Graham, R. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, XLV, No 9 1966).

[6] Ullman, J. D. Polynomial complete scheduling problems. In *Proceedings of the Proceedings of the fourth ACM symposium on Operating system principles* (1973).

[7] Graham, R., E. Lawler, E. Lenstra, A. Rinnooy Kan Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey. *Annals of Discrete Mathematics*, 51979), 287-326.

[8] Brucker, P. *Scheduling algorithms*. Springer, 2007.

[9] Sinnen, O. *Task scheduling for parallel systems*. Wiley-Interscience, Hoboken, N.J., 2007.

[10] Sih, G. C. and Lee, E. A. A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures. *Ieee Transactions on Parallel and Distributed Systems*, 4, 2 (Feb 1993), 175-187.

[11] Kwok, Y. K. and Ahmad, I. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *Ieee Transactions on Parallel and Distributed Systems*, 7, 5 (May 1996), 506-521.

[12] Gillies, D. W. and Liu, J. W. S. Scheduling Tasks with and/or Precedence Constraints. *Siam Journal on Computing*, 24, 4 (Aug 1995), 797-810.

[13] Cormen, T. H., Leiserson, C. E. and Rivest, R. L. *Introduction to algorithms*. MIT Press ;McGraw-Hill, Cambridge, Mass., 1990.

[14] Tomasulo, R. M. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 1967, 25-33.

[15] Hennessy, J. L., Patterson, D. A. and Arpaci-Dusseau, A. C. *Computer architecture : a quantitative approach*. Morgan Kaufmann, Amsterdam ; Boston, 2007.

[16] *NVIDIA CUDA™ Programming Guide Version 3.0*.

[17] Nickolls, J., Buck, I., Garland, M. and Skadron, K. Scalable parallel programming with CUDA. In *Proceedings of the ACM SIGGRAPH 2008 classes* (Los Angeles, California, 2008).

[18] *Intel® Concurrent Collections*, 2012.