A Tale of Three Runtimes

Nicolas Vasilache, Muthu Baskaran, Tom Henretty, Benoit Meister,
M. Harper Langston, Sanket Tavarageri, Richard Lethin
Reservoir Labs Inc. 632 Broadway, New York, NY, 10012[1]
January 30, 2013

arXiv:1409.1914v1 [cs.DC] 5 Sep 2014

Abstract

This contribution discusses the automatic generation of event-driven, tuple-space based programs for task-oriented execution models from a sequential C specification. We developed a hierarchical mapping solution using auto-parallelizing compiler technology to target three different runtimes relying on event-driven tasks (EDTs). Our solution benefits from the important observation that loop types encode short, transitive relations among EDTs that are compact and efficiently evaluated at runtime. In this context, permutable loops are of particular importance as they translate immediately into conservative point-to-point synchronizations of distance 1. Our solution generates calls into a runtime-agnostic C++ layer, which we have retargeted to Intel's Concurrent Collections (CnC), ETI's SWARM, and the Open Community Runtime (OCR). Experience with other runtime systems motivates our introduction of support for hierarchical async-finishes in CnC. Experimental data is provided to show the benefit of automatically generated code for EDT-based runtimes as well as comparisons across runtimes.

1 Introduction

Hardware scaling considerations associated with the quest for exascale and extreme scale computing are driving system designers to consider event-driven-task (EDT)-oriented execution models for executing on deep hardware hierarchies. This paper describes a method for the automatic generation of optimized event-driven, tuple-space-based programs from a sequential C specification. The work builds on the Reservoir Labs R-Stream compiler [MVW+11, LLM+08] and is implemented as an experimental feature in the R-Stream compiler.

We have developed a hierarchical mapping solution using auto-parallelizing compiler technology to target three different runtimes based on EDTs. Our

[1] This material is based upon work supported by the Department of Energy under Award Number(s) No. DE-SC0008717.

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

This report describes Patent Pending Technology.

solution consists of (1) a mapping strategy with selective trade-offs between parallelism and locality to extract fine-grained EDTs, and (2) a retargetable runtime API that captures common aspects of the EDT programming model and uniformizes translation, porting, and comparisons between runtimes. At a high level, complex loop nest restructuring transformations are applied to construct a logical tree representation of the program. This representation is mapped to a tree of EDTs. Each EDT is associated with a unique (id, tag tuple) pair in the generated program. Our solution generates calls into a runtime-agnostic C++ layer (RAL), which we successfully retargeted to Intel's CnC, ETI's SWARM, and the Open Community Runtime (OCR). We provide different experiments to characterize the performance of our auto-generated code and targeted runtimes.

2 A Motivating Example

```
for (t1=-1; t1<=⌊(pT-2)/8⌋; t1++) {
 lbp=max(⌈t1/2⌉,⌈(16*t1-pT+3)/16⌉); ubp=min(⌊(pT+pN-2)/16⌋,⌊(8*t1+pN+7)/16⌋);
 doall (t2=lbp; t2<=ubp; t2++) {          // par.
  for (t3=max(0,⌈(t1-1)/2⌉,⌈(16*t2-pN-14)/16⌉);
   t3<=min(⌊(pT+pN-2)/16⌋,⌊(8*t1+pN+15)/16⌋,⌊(16*t2+pN+14)/16⌋); t3++) {
   // for t4, t5, t6, t7, t8 loops omitted
   if (t5 % 2 == 0) S1(t5, t6, t7, t8);
   if (t5 % 2 == 1) S2(t5, t6, t7, t8);
}}}
```

(a) OpenMP parallel version (8x16x16x128)

```
 for (t1=⌈(-pN-15)/16⌉;t1<=⌊(pT-3)/16⌋;t1++){ //perm
   for (t2=max(t1,-t1-1);
    t2<=min(⌊(-8*t1+pT-2)/8⌋,⌊(8*t1+pN+7)/8⌋,⌊(pT+pN-2)/16⌋);t2++){ //perm
     for (t3=max(0,⌈(t1+t2-1)/2⌉,⌈(16*t2-pN-14)/16⌉);
      t3<=min(⌊(pT+pN-2)/16⌋,⌊(16*t2+pN+14)/16⌋,⌊(8*t1+8*t2+pN+15)/16⌋);t3++){ //perm
      // for t4, t5, t6, t7, t8 loops omitted
      if (t5 % 2 == 0) S1(t5, t6, t7, t8);
      if (t5 % 2 == 1) S2(t5, t6, t7, t8);
 }}}
```

(b) EDT-ready permutable version (8x16x16x128)

Figure 1: Diamond-tiled Heat-3D Kernel (parallel and permutable versions)

We begin by presenting a motivating example based on state-of-the-art techniques applied to the heat equation kernel. Previous contributions have highlighted the inherent scalability limitations of implicit stencils with time-tiling schemes [KBB+07]. Intuitively, the pipelined start-up cost needed to activate enough tasks to feed N processors grows linearly with the number of processors. When N is large, this translates into a sequential bottleneck. On large machines, the traditional solution is to perform domain decomposition and to exchange ghost regions in a near-neighbor collective communication step. A viable alternative for both scalable parallelism and locality is to use diamond tiling

on explicit stencils [BPB12]. Still, even in the absence of obvious ill-balanced wavefront synchronizations, significant load-balancing inefficiencies may remain. For the purpose of illustration, we begin with the pseudo-code in Figure 1; this code represents the outermost loops of a diamond-tiled, 3D heat equation solver using an explicit 3D Jacobi time step. In this version, `S1` reads an array `A` and writes an array `B`, whereas `S2` conversely reads array `B` and writes array `A`. Experiments on the code in Figure 1(a) have been recently published and demonstrate significant advantages over a domain specific language [TCK+11]. The version in Figure 1(b) corresponds to the same code without the "tile-skewing" transformation.[2] As a consequence, the loop types of the inter-task loops are all "permutable" and contain only forward dependences [Gri04]. We then apply the distributed dependence evaluation portion of our framework described in section 4.7. To this end, we manually insert the above code into a CnC skeleton and perform experiments on a 2-socket, 6-core-per-socket Intel Xeon E5-2620 running at 2.00 GHz. This port required minimal time once the infrastructure was established. We can draw a few conclusions from these first performance

Version / Procs	1	2	3	4	6	8	12
OpenMP	14.90	8.01	6.09	4.44	3.36	**2.86**	3.16
CnC	13.71	7.03	4.82	3.73	2.75	**1.98**	2.16
OpenMP-N	14.83	7.96	5.87	4.18	3.05	2.40	**2.31**
CnC-N	13.79	6.85	4.59	3.48	2.44	1.89	**1.42**

Figure 2: Diamond tiling [BPB12] OpenMP vs CnC

numbers. First, there is a non-negligible benefit in single-thread mode due to reduced complexity of the loop control flow (we avoid the loop-skewing transformation which has impact down to the innermost loop). Improved single-thread performance of the RAL over OpenMP is a positive side-effect of our solution. Still, there is single-thread cost to parallelism: the base sequential runtime of the tiled code in figure 1(b) is 12.74s. Second, even if diamond-tiled code does not suffer pipeline parallel overhead, there are still benefits to be gained from load-balancing effects. Third, placement of computations and data is crucial and will be an important focus of future research. In this experiment, we pinned threads to cores and performed round-robin pinning of pages to sockets with the `libnuma` library. Rows OpenMP-N and CnC-N demonstrate the effect of NUMA pinning with still an approximate 40% socket miss rate. Pinning remains uncontrolled in rows OpenMP and CnC. More controlled data placements are expected to produce better results and will be needed as we progress to many sockets and longer off-core latencies.

[2] Time-skewing creates the OpenMP parallel loop. Since diamond tiling is used, the loop is well-balanced across multiple $t1$ iterations.

3 Position of the Problem

As projected by [KBC+08], power limits in the machine room are one of the principal constraints in reaching exascale. The strategy of relying solely on commodity server chips and accelerators, as is the case for petascale computing, will result in infeasible levels of power. Consequently, new processor and system architectures are being investigated and designed for exascale. These new architectures will have new programming and execution models.

EDT execution models reincarnate [MS68] old ideas of dataflow [Den74] and macro-dataflow [SGS+93]. What is driving this reincarnation of these old dataflow ideas is that the new exascale hardware is expected to be powered with supply voltages that are near the transistor threshold voltage [KAH+12]. Lowering supply voltage produces a quadratic improvement in power efficiency of computing devices with only a linear slowdown in throughput. Consequently it is possible to get improved power utilization *as long as an increase in parallelism can be found to offset the linear slowdown.* Of course, pure dataflow has the advantage of expressing the absolute maximum amount of parallelism to the level of individual operations, but that comes with too much overhead and does not solve the scheduling problem [CA88].

Macro-dataflow groups individual operations into larger tiles and provides scheduling algorithms with provably optimal properties [BJK+96], with the possibility of an increase in parallelism by virtue of it more accurately representing the minimal set of synchronizations, compared to the current dominant execution models utilizing bulk synchronization.

Another important consequence of lowering the supply voltage near threshold is that variations in device performance are exacerbated. Thus, beyond any intrinsic imbalance from the application itself, the hardware will be create imbalance. This means that mechanisms for dynamic load balancing increase in importance for near threshold computing (NTC).

To address this, exascale architects envision that additional degrees of parallelism must be found to overprovision the execution units to aid in load balancing [IHBZ10] and to hide remote access latencies. This overprovisioning effect is already mainstream in the GPGPU world, and in particular in CUDA, where a user specifies more parallelism than can be exploited at any given time for the purpose of hiding latencies. In the CUDA programming model, this overprovisioning still relies on the bulk-synchronous model at the device level but has begun shifting to a task-based model at the CPU-to-device level.

Another aspect of envisioned exascale architectures is that the hierarchy of the hardware, which is currently at 3-5 levels (core, socket, chassis, rack...) will extend to 10 or more levels (3-4 levels on chip, and another 6+ levels of system packaging). In current systems, execution models address these levels through loop blocking or tiling [LRW91], and stacked heterogeneous execution models (e.g., MPI+OpenMP). Such approaches will become cumbersome (program size, etc.) with the envisioned 10+ levels. More compact representations that are naturally recursive [YAK00] or cache oblivious [FLPR99] are potentially better.

Some of these tradeoffs are described in one paper describing the Intel Run-

nemede exascale research architecture [Car13]; the OCR [tea] is designed with Runnemede as one of the hardware targets it should run well on.

Traditional approaches to parallelism require the programmer to express it explicitly in the form of communicating sequential processes. The fork-join model [Boa08] and the bulk-synchronous model [PSP98] are well-established methodologies for shared and distributed memory systems respectively.

The EDT model supports the combination of different styles of parallelism (data, task, pipeline). At a very high-level, the EDT program expresses computation tasks which can: (1)produce and consume data, (2) produce and consume control events, (3) wait for data and events, and (4) produce or cancel other tasks. Dependences between tasks must be declared to the runtime, which keeps distributed queues of ready tasks (i.e., whose dependences have all been met) and decides where and when to schedule tasks. Work-stealing is used for load-balancing purposes [BJK+96]. Specifying tasks and dependences satisfied at runtime is common to CnC [Inta], OCR, SWARM [Intb] and other EDT runtimes. Unfortunately, few programs using these constructs are available to the community. Also, it is impractical to expect programmers to write directly in EDT form; the expression of explicit dependences is cumbersome, requiring a significant expansion in the number of lines of code, and opaque to visual inspection and debugging. Direct EDT programming might be done in limited circumstances by some hero or uber programmers, or in evaluation and experimentation with this execution model.

We describe a solution to automate the synthesis of EDT codes from simple sequential codes.

A viable transformation system for this purpose is based on the polyhedral model. The transformation system proposed by [GVB+06] allows very intricate transformation compositions, but its applicability is typically limited by (among others) static dependence analysis. Solutions have been devised for expanding the scope of analyzable codes by (1) computing inter-procedural over- and under-approximations [CI95], which present a conservative abstraction to the polyhedral toolchain, and by (2) introducing more general predicates evaluated at runtime through fuzzy-array dataflow analysis [CBF95]. In practice, conservative solutions mix well with the polyhedral toolchain through a stubbing (a.k.a. blackboxing) mechanism and parallelism can be expressed across irregular code regions. Unfortunately, this is not sufficient as the decision to parallelize remains an all-or-nothing compile-time decision performed at the granularity of the loop. In contrast, EDT-based runtimes allow the expression of fine-grain parallelism down to the level of the individual instruction (overhead permitting). Recently, new techniques, which allow for performing speculative and runtime parallelization using the expressiveness of the polyhedral model, have emerged [JCP+12].

[OR12] make a very strong case that a dependence analysis based on a DAG of linear-memory array descriptors can generate lightweight and sufficient runtime predicates to enable adaptive runtime parallelism. This requires runtime evaluation of predicates and results in significant speedups on benchmarks with difficult dependence structures. In their work, parallelism is still exploited

in a fork-join model via the generation of OpenMP annotations. One of the goals of task-based runtimes is to go beyond the fork-join [Boa08] and the bulk-synchronous [PSP98] models.

4 Approach

This work is based on an automatic approach to program analysis and transformation. From an analyzable sequential C specification, our algorithm performs conversion to our intermediate representation and goes through the following steps:

- Instance-wise dependence analysis with extensions to support encapsulated non-affine control-flow hidden within summary operations (a.k.a. blackboxes);
- Scheduling to optimize a trade-off between parallelism, locality and other metrics in the program, using an algorithm [LVML09], which extends the work of [BHRS08];
- Non-orthogonal tiling of imperfectly nested loops with a heuristic, which balances a model of data reuse, cache sizes and performance of streaming prefetches;
- EDT formation from a tree representation of the tiled program;
- Generation of dependences between EDTs;
- Code generation to a C++ runtime-agnostic layer (RAL).

The purpose of the RAL is to easily adapt to different runtimes.

4.1 Notations

Our intermediate representation is based on a hierarchical dependence graph. The nodes of our graph are statements, which represent operations grouped together in our internal representation. The unit of program analysis and transformation is a statement; a statement S can be simple or arbitrarily complex (i.e., an external precompiled object), as long as it can be approximated conservatively. The edges of our graph are dependences as defined below. An iteration domain for S, D^S, is an ordered multi-dimensional set of iterations. An instance of an iteration is written i_S. The (lexicographic) order relation between iterations i and j is defined by $i \ll j$ iff i occurs before j in the program. By introducing y, the symbolic constant parameters of the program, an iteration domain is the set $\{i_S \in D^S(y)\}$. Operations to manipulate domains and their inverse include projections to extract information along a subdomain; image by a function to transform a domain into another domain; intersection to construct the iterations that are common to a list of domains; and index-set splitting to break a domain into disjoint pieces. Even at compile time, exact projection

operations are often prohibitively expensive, and we discuss the implications in section 4.4.

A scheduling function Θ^S is a linear affine function that partially reorders the iterations of S in time. The order $\ll$ extends to time after scheduling is applied. In this context, a dependence $(T \to S)$ is a relation[3] between the set of iterations of S and T. It conveys the information that some iteration $i^T \in D^T(y)$ depends on $i^S \in D^S(y)$ (i.e., they access the same memory location by application of a memory reference) and that $i^S \ll i^T$ in the original program. We write the set relation $\{(i^T, i^S) \in \mathcal{R}_{T \to S}(y))\}$ or $\mathcal{R}_{T \to S}(y)$ to refer to the specific iterations of T and S that take part in the dependence. The multigraph of statement nodes and dependence edges is referred to as the **generalized dependence graph** (GDG). We write GDG=(V,E), the set of vertices and edges in the graph respectively.

4.2 Scheduling

The R-Stream scheduler is a state-of-the-art parallelization tool, which optimizes parallelism and locality in sequences of imperfectly nested loops. An example of an earlier and more easily digestible algorithm is presented in [BHRS08]. Optimization is obtained by unifying the tilability conditions expressed by [IT88] with scheduling techniques introduced by [Fea92]. We briefly review the affine scheduling formulation. The input of the affine scheduling problem is a GDG. Following the standard conventions, ϕ_S is used to denote a 1-dimensional affine schedule for statement S. For each edge in the GDG, [BHRS08] writes:

$$\delta(y) \geq \phi_T(i_T, y) - \phi_S(i_S, y) \geq 0, \qquad (i_T, i_S) \in \mathcal{R}_{T \to S}(y) \tag{1}$$

By combining all the dependences of the program, one forms a feasible linear space that can be subject to various optimization problems. The parametric affine form $\delta(y)$ can be interpreted as the maximal dependence distance between any two schedules. In particular, if $\delta(y)$ can be minimized to 0, then the solution ϕ is communication-free and is thus *parallel*. Similarly, if $\delta(y)$ can be minimized to a positive constant c, then only local communication is needed and broadcast can be eliminated. Bondhugula's iterative algorithm, allows finding independent solutions that are valid for the same set of dependence edges. This implies the induced loops are permutable.

Bondhugula's algorithm is shown in Figure 3. A brief summary of the steps in the algorithm is in order:

- In step (2), we find as many solutions to (1) as possible, while minimizing the coefficients to $\delta(y)$. As the algorithm proceeds, edges from E are removed. In each iteration of step (2), we only consider the edges in E that are remaining.
- Steps (3)-(5) are triggered if step (2) fails to find a solution. If so, dependences from different SCCs in the GDG are cut. This has the effect

[3] We use the notation $T \to S$ to express that T depends on S

1: **repeat**
2: Find as many independent solutions to (1) as possible.
3: **if** no solutions were found **then**
4: Cut dependences between SCCs in the GDG.
5: **end if**
6: Remove from E all edges satisfied in step (2).
7: **until** $E = \emptyset$;

Figure 3: Bondhugula's algorithm.

of fissioning the loops in different SCCs. This edge cutting can be done incrementally.

- In step (6), all satisfied edges are removed from the graph. An edge e is *satisfied* by a solution ϕ iff

$$\phi_{\mathsf{s}(e)}(i, y) - \phi_{\mathsf{t}(e)}(j, y) \geq 1, (i, j) \in \mathcal{R}_{T \to S}(y)$$

Note that this edge removal step is not performed in step (2). This ensures that all solutions generated in step (2) are permutable, i.e. valid for the same set of dependence edges. The ability to extract general permutable loops is essential to scalable generation of EDTs.

4.3 Scalability

The expressiveness of the compositions of transformations allowed by the polyhedral model is also the source of scalability challenges. Traditionally, scalability challenges come in two flavors: scheduling computations and generating code for tiled loops. In the context of EDTs, a third challenge appears: the tractability of dependence computations between EDTs. The scalability of scheduling problem is out of the scope of this paper. The scalability of code generation of tiled loops is of direct relevance to our work as we aim to generate EDTs for multiple levels of hardware hierarchy. The example pseudocode sketched in section 2 gives a first insight into the complexities involved when simultaneously optimizing parallelism and locality in a polyhedral toolchain. The constraints introduced for tiling can be expressed with simple loops that capture the constraints: $8t_1 \leq t_5 \leq 8t_1 + 7$, $16t_2 \leq t_6 \leq 16t_2 + 15$, $16t_3 \leq t_7 \leq 16t_3 + 15$ and $128t_4 \leq t_8 \leq 128t_4 + 127$. However, the complete resulting code is significantly more complex and involves multiple min/max expressions as well as ceil and floor divisions. The advantage is that optimal control-flow, in the number of loop iterations executed, is guaranteed [Bas04b]. This is a common tradeoff between the complexity of the representation (which is worst-case exponential in the number of induction variables and symbolic parameters) and the quality of the generated control-flow. In practice, tiling introduces additional induction variables at compile time to specify different inter-tile and intra-tile iteration

ordering and it is rare to see more than two levels of tiling generated by pure polyhedral solutions. Parameterized tiling has been proposed as a solution to the scalability limitation in the polyhedral model for multiple levels of hierarchy [RKRS07, HBB+09]. Unfortunately, these techniques are non-linear and don't allow further optimizations to be carried within the model. In particular, special considerations need to be taken into account to allow parallelism and even so, only a single level of imbalanced, wavefront-style parallelism is supported [BHT+10]. A key innovation of this paper is based on the insight that *the availability of EDT based runtimes changes this limitation in the context of parametric tiling.*

This work makes use of parameterized tiling. A tile in the polyhedral model we propose to allow imperfect control-flow (which may exhibit empty iterations) in order to achieve a more scalable representation and the ability to generate multi-level code. Techniques for symbolic Fourier-Motzkin elimination have been developed to reduce the potential overhead of empty iterations [BHT+10] Symbolic runtime Fourier-Motzkin elimination is also used by [OR12].

Lastly, an EDT-specific challenge lies in the tractable computation of dependences at compile-time and the overhead of their exploitation at runtime. This is discussed in the next section.

4.4 On Scalable Dependence Computation Between EDTs

The runtimes we target all require the programmer to specify dependences between EDTs to constrain the order of execution for correctness purposes. Dependence relations are exploited by the runtime to determine when a task is ready and may be scheduled for execution. An EDT-specific challenge lies in the tractable computation of dependences at compile-time and the overhead of their exploitation at runtime.

The requirements for dependence relations between EDTs is significantly different than for dependence analysis between statements. Dependence analysis is only concerned with original program order of statements and can be captured by the set.

$$\mathcal{R}_{T\to S}(y) = \left\{(i^S, i^T) \in D^S \times D^T \mid i^S \ll i^T, M_S[i^S] = M_T[i^T]\right\},$$

where M_S and M_T are memory access functions (typically Read-After-Write affine indices in the same array). Array dataflow analysis [Fea91] goes a step further and takes into account all possible interleaved writes to only keep the true producer-consumer dependences. A dataflow dependence is then written:

$$\mathcal{F}_{T\to S}(y) = \mathcal{R}_{T\to S}(y) - \left\{\bigcup_W \prod_{T\times S} \left(\mathcal{R}_{T\to W}(y) \times \mathcal{R}_{W\to S}(y)\right)\right\},$$

where $\prod$ is the projection operator from $T \times W \times S$ to $T \times S$.[4] In the case of

[4] In Feautrier, a parametric Integer Linear Programming (ILP) problem is solved and there is no need for a projection operator. The formulation with set differences and the projector operator merely simplifies the exposition of the problem.

dependences between EDTs these additional points must be taken into account:

- ordering must be computed on the transformed schedule which comprises tiling transformations, possibly at multiple levels,
- multiple instances of a statement belong to the same tile. This is a projection operation that cannot be avoided when computing dependences exactly,
- by virtue of exploiting parallelism, the "last-write" information becomes dynamic and introduces the need for sets of dependence relations.

A possible way to automatically specify these relations is to exactly compute dependences between tasks at compile-time based on producer-consumer relationships as has been proposed earlier [BVB+09]. In this context, the following issues arise:
1) Dependences may be redundant. A straightforward dependence algorithm considers producer-consumer relations on accesses to memory locations. In the context of EDTs without special treatment to prune redundancies, all these dependences would be generated translating into a high runtime overhead.
2) Perfectly pruning dependences statically requires the static computation of the "last-write" whose general solution is a Quasi-Affine Selection Tree (QUAST) [Fea88]. Computing this information exactly is often very expensive on original input programs. After scheduling and tiling, the complexity of the "last-write" computation is further increased by the application of a projection operator (multiple statement instances belong to the same tile instance). An alternative is to compute "covered dependences" on the original program [PW92] before scheduling and tiling and update them as scheduling and tiling are applied. This approach still would require projection and redundancy removal. Additionally, it is not expected to scale when multiple levels of tiling are involved.

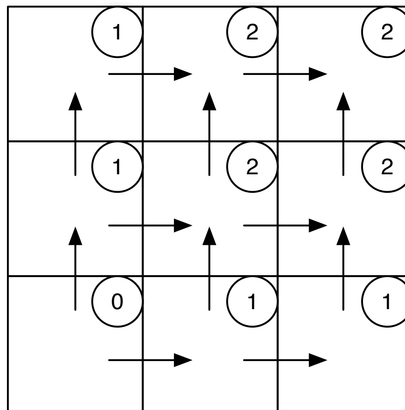

Figure 4: EDT antecedents graph

3) Dependence relations between tasks are generally non-convex, arising from the projection of the dependence relation on a subspace. The projection operator is non-convex. In figure 4, consider the possible dependence paths between the origin $(i, j) = (0, 0)$ and $(i, j) = (1, 1)$. These correspond to the number of paths of Manhattan distance 2 between these points on a uniform 2-D grid. In particular, task $(i, j) = (0, 0)$ has no antecedents and can start immediately whereas $(i, j) = (1, 1)$ has 2 antecedents. Task (i, j) has $i \cdot j$ redundant dependences (i.e. the "volume" of the $i \times j$ region) which reduces to 0, 1 or 2 transitively unique dependences. In general, for one level of hierarchy, the number of these dependences varies whether the task coordinates sit on the edge vertex, edge line or interior of the 2D task space. [BVB+09] handle this case by creating a centralized, sequential loop that scans the Cartesian product of iteration spaces of source and destination EDTs for each single dependence. This mechanism incurs high overhead. In their experiments, tile sizes are kept larger to amortize this overhead. However, if overprovisioning is to be achieved, EDTs have to be much smaller and the overhead of having smaller EDTs is very high in the work of Baskaran et al. We ran experiments with the implementation of Baskaran et al. to confirm our perceived limitations of their approach. Especially with benchmarks where the loop nest is deeper, when the number of EDTs is increased with smaller tile sizes, the task graph with dependences incurs a huge memory and performance overhead.

For all the reasons previously mentioned, we developed a new algorithm based on loop properties rather than explicitly computing all possible dependences between tasks.

4.5 Tree Representation and EDT Formation

After scheduling and tiling, the transformed program is represented as a tree of imperfectly nested loops, similar to an abstract syntax tree (AST). Two main differences differences arise between a traditional AST and a tree of loops in our model:

- our representation is oblivious to the effects of loop transformations (among which are peeling, loop shifting, parameter versioning and index-set splitting). This is a particular strength of compositions of transformations in the model we adopted. Code generation deals with the intricacies of reforming the control-flow-efficient transformed loop nests [Bas04a],
- subsequent loop transformations are further composable and preserve the independence of the representation with respect to complex control flow.

The tree structure is characterized by the integer "beta vector" that specifies relative nesting of statement [GVB+06]:

- statements have identical first d beta component if and only if they are nested under d common loops. The bounds of the loops may be completely different for each statement,

- as soon as the beta component differ, the loops are distributed, the order of the loops being consistent with the order of the beta component

Each node in the tree therefore corresponds to a loop and has a loop type associated with it. To uniformize the EDT extraction algorithm, we also introduce a root node in the tree that does not correspond to any loop but is the antecedent of all nodes. The algorithm in Figure 5 performs a breadth-first traversal on the tree structure induced by the beta vectors and marks nodes. The algorithm forms sequences of perfectly nested consecutive loops with compatible types for dependence inference purposes (see section 4.6). In particular, permutable loops of the same band can be mixed with parallel loops. However, permutable loops belonging to different bands cannot be mixed in the current implementation.

```
1.  Mark the root node of the tree
2:  repeat
3:      Let N be the next node in the BFS traversal
4:      if N is at tile granularity or
5:          N is user-provided then mark N
6:      else if N is sequential then mark N
7:      else if N is has siblings then mark N
8:      else if N is permutable and
9:          N is in a different band than parent(N) and
10:         parent(N) is not marked
11:     then
12:         mark N
13:     end if
14: until tree traversed;
```

Figure 5: EDT formation algorithm

At the moment, we support two strategies. The default approach consists of stopping traversal when the granularity of a tile is reached. This creates EDTs at the granularity of tiles. The second strategy lets the user decide which nodes should be marked and ignores tile granularities. The algorithm introduces the remaining nodes necessary to accommodate changes in permutable bands as well as sequential loops and imperfectly nested loops that would require too many dependences and that we handle in a special way (see section 4.6).

Once the tree is marked, one compile-time EDT is formed for each marked *non-root* node. This proceeds as follows:

1. pick a new unique id for the current EDT,
2. determine the start and stop level for this EDT. The start level is the level of the first marked ancestor, the stop level is the level of the node,
3. filter the statements nested below the node and attach them to the current EDT,

4. form the iteration domain of the EDT as the union of the domains of all its statements.

One ends up with a tree of EDTs, **whose coordinates in a multi-dimensional tag space are uniquely determined by loops** $[0, stop]$. Coordinates $[0, start)$ are received from the parent EDT, coordinates $[start, stop]$ are determined locally from loop expressions. The combination of an EDT id and its coordinates uniquely identify each EDT instance.

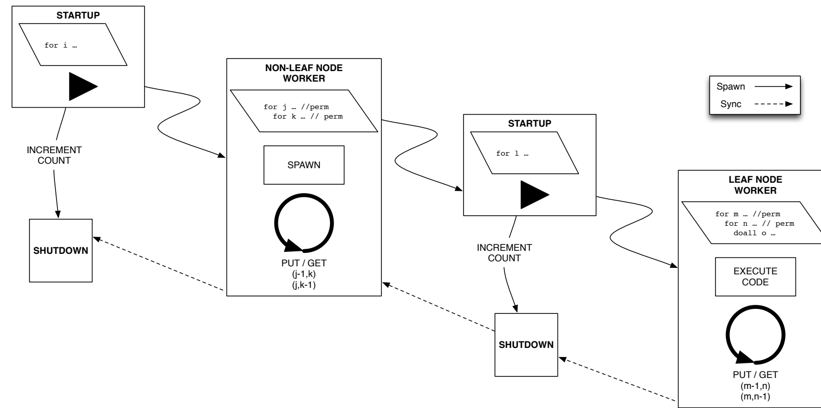

Figure 6: Global organization of EDTs spawning and synchronization.

The code generation process creates three different runtime EDTs for each compile-time EDT:

1. a STARTUP EDT spawns WORKER EDTs asynchronously and sets up a counter to determine when a subgroup of WORKER EDTs is complete,
2. a WORKER EDT either further decomposes into additional levels of EDTs or performs work if it is a leaf in the tree, and
3. a SHUTDOWN EDT acts as a synchronization point for tasks generated in step 1.

Vertically, across steps 1 - 3, counting dependences capture hierarchical async-finish relations and are used to implement hierarchical synchronization points. Horizontally, within step 2, point-to-point dependences provide load-balancing support. Figure 6 illustrates the organization of spawning and synchronizations across the three types of EDTs. This overall structure is further described in section 4.8.

4.6 Dependence Specification With Loop Types Information

When performing dependence analysis, one obtains a direct relation between iterations in the source and target domains. These relations should be cheap

to determine at compile time (i.e. not require projections), should evaluate at runtime in a distributed, asynchronous mode and should not require iterating over a high-dimensional loop nest at runtime. To avoid iterations over a high-dimensional loop nest, we adopt a decentralized view of the problem using a *get-centric* approach in which an EDT queries its predecessors whether they have finished executing. This minimizes the number of *put*s in a concurrent hash table, which are notoriously more expensive than gets.

Individual dependence relations between statements are generally non-invertible. Consider the relation $[i, i] \rightarrow [i, j]$ [5]. Forming the inverse relation requires a projection and gives rise to (many) dependences at runtime. A key insight of this work is that although individual dependence relations between statements may require projections, *aggregate dependence relations* between EDTs may not. Loop types exhibited by the scheduler and described in this section allow the scalable computation of these aggregate dependences.

Parallel loops are the simplest type of dependence relation in the program: they carry no dependence. As a consequence, no special conditional needs to be evaluated at runtime.

A permutable band of loops over induction variables $(i_1, \ldots, i_n)$ has only forward dependences. These can always be expressed conservatively by the set of n invertible relations:

$$\{[i_1, i_2, ..., i_n] + e_k \rightarrow [i_1, i_2, ..., i_n], k \in [1, n]\},$$

where e_k is the canonical unit vector in dimension k. In order to infer dependences for a nest of permutable loops, each task determines from its coordinates in the task space whether it is a boundary task or an interior task and which other tasks it depends on. For the purpose of illustration, we consider 3D tasks (i.e., loops `(i, j, k)` are loops across tasks or inter-task loops). The code shown in Figure 8 exploits dependences of distance 1 along each dimension (i.e., Boolean expressions are formed plugging `i-1`, `j-1`, and `k-1` into the expression of the loop bounds). For each permutable loop dimension, we introduce a condition to determine whether the antecedent of the task along that dimension is part of the interior of the inter-task iteration space. When the condition evaluates to `true`, the task must wait (i.e. get) for its antecedent to complete. Due to space considerations we only show the interior_1 computation; interior_2 and interior_3 are omitted.

Sequential is the most restrictive type for loops. It imposes a fully specified execution order on the current loop with respect to *any loop nested below it.* To visualize these effects, consider the code in figure 7 where the function f reads a portion of array A such that the dependence structure is `(seq,doall,seq,doall)`. Suppose that a task has the granularity of a single `(t,i,j,k)` iteration of the innermost statement. The dependence semantic from loop `t` is that *any* task `(t,i,j,k)` depends on *all* of its antecedents `(t-1,*,*,*)`. Similarly from loop `j`, *any* task `(t,i,j,k)` depends on *all* of its antecedents `(t,i,j-1,*)`. If all dependence relations were to be exposed to the

[5] With our convention this reads: iteration $[i, i]$ depends on iteration $[i, j]$

```
for (t=1; t<T; t++) {
  doall (i=1; i<N-1; i++) {
    for (j=1; j<N-1; j++) {
      doall (k=1; k<N-1; k++) {
        A[t][i][j][k] = f(A);
}}}}
```

Figure 7: Effect of sequential loops

runtime, the `t` loop would require N^6 dependences which is prohibitive. The naive solution gets even worse when multiple levels of hierarchy are introduced: if each loop `i`, `j`, `k` were split in 2 equal parts, each task's granularity would be reduced by a factor of 2^3. The naive approach would yield an increase in number of dependences by a factor of 2^6 and is clearly not a viable alternative. A simple idea consists in generating a 1D fan-in/fan-out task similar to

```
// Pseudo-code to spawn tasks asynchronously
for (i=⌈(-N-15)/16⌉;i<=⌊(T-3)/16⌋;i++) {
 for (j=max(i,-i-1);
      j<=min(⌊(-8*i+T-2)/8⌋,⌊(8*i+N+7)/8⌋,⌊(T+N-2)/16⌋);j++) {
  for (k=max(0,⌈(i+j-1)/2⌉,⌈(16*j-N-14)/16⌉);
    k<=min(⌊(T+N-2)/16⌋,⌊(16*j+N+14)/16⌋,⌊(8*i+8*j+N+15)/16⌋);k++) {
     spawn_task_async(tag(i, j, k));

// Input: tag t
int i=t.i, j=t.j, k=t.k;
// Use enclosing loops' upper and lower bounds to
// determine whether the task sits on a boundary
bool interior_1 =
  (( i-1 >= ⌈(-N-15)/16⌉) && ( i-1 <=  ⌊(T-3)/16⌋))                    &&
  ((j >= ( i-1,-i))                                                    &&
   (j <= min(⌊(-8*( i-1)+T-2)/8⌋,⌊(8*( i-1)+N+7)/8⌋),⌊(T+N-2)/16⌋)) &&
  ((k >= max(0,⌈(( i-1)+j-1)/2⌉,⌈(16*j-N-14)/16⌉))                    &&
   (k <= min(⌊(T+N-2)/16⌋,⌊(16*j+N+14)/16⌋,⌊(8*( i-1)+8*j+N+15)/16⌋)));
if (interior_1) wait_task(tag( i-1, j, k));  // a.k.a get
//   EDT body
signal_task_finished(tag(i, j, k));         // a.k.a put
```

Figure 8: Antecedent determination in permutable loop nest

a `tbb::empty_task`. This has the effect of reducing the "Cartesian product" effect of dependence relations. The dependence semantic becomes: *any* task `(t,i,j,k)` depends on `sync(t)` and `sync(t)` depends on *all* tasks `(t-1,i,j,k)`. The number of dependences reduces to $2N^3$ which can still be too high. The approach we exploit for sequential loops is based on *hierarchical separation of concerns*: a sequential loop generates an additional level of hierarchy in the task graph, effectively acting as a tbb::spawn_root_and_wait. To accommodate this separation of concerns, we generate three different runtime EDTs for each compile time EDT.

Towards More Flexible Semantics Without deeper compiler introspection, the loop semantics we described may sometimes be too conservative. In figure 9, we provide two toy examples in which conservativeness reduces parallelism. In

```
for (t=1; t<T; t++) {
  doall (i=1; i<N-1; i++) {
    A[t+1][i] = C0*A[t-1][i];
}}}}
```

```
for (t=1; t<T; t++) {
  doall (i=1; i<N-1; i++) {
     A[t][i] = C0*A[T-t][i];
 }}}}
```

Figure 9: Conservative dependence distance (left) and index-set (right)

the first case, dependence distances of length 1 are too conservative. Dependence distances of length 2 enable twice as many tasks to be executed concurrently. More generally, for a set of dependences with constant distances, the greatest common divisor of the distances gives the maximal amount of parallelism. In the second case, the `t`-loop could be cut (by index-set-splitting) in 2 portions that do not contain any self-dependence. More generally, for a set of dependences, the transitive closure of dependence relations is needed to fully extract the maximum available parallelism [GFL99].

The two simple cases presented can be captured in a simple framework implementing the technique illustrated in this section. The first case can be dealt with by specifying a *dependence relation between inter-task indices* that generalizes the relation illustrated in figure 8 (which only considers $\{\texttt{t-1} \rightarrow \texttt{t}\}$, $\{\texttt{i-1} \rightarrow \texttt{i}\}$ and $\{\texttt{j-1} \rightarrow \texttt{j}\}$). The second case can be solved by augmenting the computations of the Boolean expressions (`interior_1`, `interior_2`, and `interior_3`) to include additional filtering conditions. In other words, *the effect of index-set-splitting* is applied on the Boolean computation only and **not on the iteration domain of the statements, as is customary**. The consequence is that the complexity of control-flow resulting from index-set splitting will not contaminate the iteration domains of the statements as well as potentially non-convex dependences (replaced by the Boolean evaluations in figure 8).

The cases discussed above are simple illustrations and multiple difficulties arise when both cases come together to provide additional parallelism. Tradeoffs will exist between the tractability of the static analysis involving the computation of transitive closures of the different sets of dependences [GFL99], scalability to deep levels of hierarchy, and impact of the information derived on the amount of parallelism in the program. It will be of paramount importance to focus efforts on cases that occur in practice in applications of relevance to Exascale.

4.7 Runtime Agnostic Layer

The RAL comprises of a set of C++ templated classes to build expressions evaluated at runtime, along with an API that our compiler targets. The API aims at being a greatest common denominator for features across runtimes. Since our compiler does not support C++ natively, we hide some of the implementation

⟨*linear-expr*⟩ ::= ⟨*number*⟩
| ⟨*induction-term*⟩
| ⟨*parameter*⟩
| ⟨*number*⟩ '`*` ' ⟨*linear-expr*⟩
| ⟨*linear-expr*⟩ '`+` ' ⟨*linear-expr*⟩
| ⟨*linear-expr*⟩ '`-` ' ⟨*linear-expr*⟩

⟨*expr*⟩ ::= ⟨*linear-expr*⟩
| '`MIN(`' ⟨*expr*⟩ '`,` ' ⟨*expr*⟩ '`)`'
| '`MAX(`' ⟨*expr*⟩ '`,` ' ⟨*expr*⟩ '`)`'
| '`CEIL(`' ⟨*linear-expr*⟩ '`,` ' ⟨*number*⟩ '`)`'
| '`FLOOR(`' ⟨*linear-expr*⟩ '`,` ' ⟨*number*⟩ '`)`'
| '`SHIFTL(`' ⟨*linear-expr*⟩ '`,` ' ⟨*number*⟩ '`)`'
| '`SHIFTR(`' ⟨*linear-expr*⟩ '`,` ' ⟨*number*⟩ '`)`'

⟨*comp-expr-list*⟩ ::= ⟨*comp-expr*⟩ '`,` ' ⟨*comp-expr-list*⟩ | ⟨*comp-expr*⟩

⟨*multi-range*⟩ ::= ⟨*range*⟩ '`;`' ⟨*multi-range*⟩ | ⟨*range*⟩

Figure 10: Range grammar

behind C macros. Note that in order to support SWARM, C macros are extensively needed. The central element is the templated TaskTag which encapsulates the tuple holding the coordinates of the EDT in the tag space. Specialized tuples for each runtime derive from this TaskTag and optionally extend it with synchronization constructs to implement async-finish. TaskTags are passed along with EDTs as function parameters in SWARM, OCR and CnC.

4.7.1 Templated Expressions

We implemented an approach based on C++ template expressions to capture complex loop expressions and dynamically evaluate inverse dependence relations.

Figure 10 encodes the multi-dimensional ranges that can be generated by our method. Operations on these ranges take tuples as input and return Booleans or other tuples. Supported operations include evaluation of the expression at a tuple, comparisons at a tuple and computations of the minimum and maximum given a tuple range (bounding box computation). These expressions are used as described below, following Figure 8. First a tuple of "terms" is created that encapsulates the induction variables (`t1`, `t2`, `t3`) and parameters (`T`, `N`). Then, templated expressions `p1`, `p2`, and `p3` are declared that capture the lower and upper bound expressions governing the iterations of the terms. Lastly, based on the lower and upper bound expression, runtime dependences are generated using a templated construct that dynamically captures the non-convex Boolean evaluations from figure 8. These expressions are oblivious to the complexity of the loop expressions, which can become a severe bottleneck in a polyhedral

IR based on dual representation of constraints and vertices. The tradeoff is the runtime overhead for constructing and evaluating the expression templates. Experiments with vtune and particular efforts in keeping this overhead low by using C++11's `constexpr` types and declaring the expressions static, show an overhead below 3% in the worst cases we encountered.

4.7.2 EDT Code Generation

We extended the CLOOG code generator [Bas04a] to walk a tree of EDTs in a recursive, top-down traversal. Each EDT is generated in its own separate file. These are later compiled independently with gcc and linked with the runtime library to produce the final executable. SHUTDOWN EDTs do not require special treatment, they always consist in the same code, only parametrized by the TASKTAG. Each STARTUP and WORKER EDT is parametrized by a start and a stop level. Until the start level, three behaviors can be observed: (1) induction variables and parameters are retrieved directly from the EDT tag by using the overloading of TaskTag::operator=, (2) loops are forcibly joined by forming the union of their domains and (3) loops are forcibly emitted as conditionals. Between the start and stop levels, the code generation process follows the normal behavior of CLOOG, separating statements and generating loops and conditionals in each subbranch of the code tree. After the stop level, behavior depends on the type of EDT:

- for a STARTUP, we generate the counting variable increment in the first loop and the spawning of WORKER EDTs in the second loop,
- for a non-leaf WORKER, we generate code to recursively spawn a STARTUP,
- for a leaf WORKER, the computations and communications corresponding to the actual work are generated.

4.7.3 Runtimes Discussion

Concurrent Collections (CnC) is a high-level coordination language that lets a domain expert programmer specify semantic dependences in her program without worrying about what runs where and when. CnC has a task-graph model that is *implicit* in the program representation by using hash tables. Intel-CnC ships with a work-stealing runtime whose default scheduler is built on top of the scheduler provided by Intel's Threading Building Blocks(TBB) [Rei07]. TBB was strongly inspired by STAPL [RAO98]. CnC uses tbb::concurrent_hashmap to implement step and item collections. A step is a C++ object that implements an execute method; it represents a scheduled unit of execution. The CnC scheduler decides at runtime which step::execute methods are called on which hardware thread and on which processor. This step::execute method takes a step tag reference and a context reference. A step becomes available when an associated step tag is put in the proper step collection. A step may perform multiple gets and puts from/to item collections. Item collections act as dataflow

dependence placeholders. By default, a CnC get is blocking. If it fails, control is given back to the scheduler which re-enqueues the step to await the corresponding tag put. Once that put occurs, the step restarts. In the worst-case scenario, each step with N dependences could do N-1 failing gets and be requeued as many times. Additionally, on a step suspension, the gets are rolled back. Performing all gets of a step before any put offers determinism guarantees [BBC+10].

ETI's SWift Adaptive Runtime Machine (SWARM) [Intb] is a low-level parallel computing framework that shares similarities with CnC. Additionally, SWARM handles resource objects and allows active messages and continuation passing style. In this work we only exploit the features that are common to CnC. SWARM is a C API that makes extensive use of pre-processor macros. In SWARM, an EDT is declared as a C macro and scheduled into the runtime by calling the swarm_schedule function. An EDT accepts a context parameter THIS and an optional parameter INPUT that come in the form of pointers. SWARM allows more complex behaviors where a parent EDT specifies a NEXT and NEXT_THIS parameter to allow chaining of multiple EDTs. SWARM also allows an EDT to bypass the scheduler and dispatch another EDT immediately using swarm_dispatch. The tagTable put and get mechanisms in SWARM are fully non-blocking. It is the responsibility of the user to handle the synchronization properly, re-queue EDTs for which all gets did not see matching puts and terminate the flow of execution for such EDTs. SWARM presents a lower-level runtime and API and allows many low level optimizations.

The Open Community Runtime (OCR [tea]) is a third runtime system based on EDTs and work-stealing principles. OCR represents the task graph explicitly and does not rely on tag hash tables. In OCR different objects can be specified as "events", whether they represent EDTs, blocks of data ("datablocks") or synchronization objects. OCR does not natively rely on a tag space. Instead, when an EDT is spawned, all the events it depends on must have already been created by the runtime and must be passed as dependence parameters to the EDT. By contrast, in CnC and SWARM, when a get is performed, the corresponding hash table entry can be viewed as a type of "synchronization future." There is effectively a race condition between the first get, the subsequent gets and the first put with a given tag. Additionally, mapping to a tag tuple to an event is necessary to create the synchronizations. Without a hash table, OCR requires the pre-allocation of a large number of synchronization events (as is demonstrated in the Cholesky example that ships with OCR). We chose to implement a *prescriber* in the OCR model to solve this race condition. Puts and gets are performed in a *tbb::concurrent_hash_map* following the CnC philosophy. [6] This PRESCRIBERstep is completely oblivious to the compiler and is fully handled by the RAL. In our targeting OCR, each WORKER EDT is dependent on a PRESCRIBER EDT which increases the total number of EDTs. Lastly, OCR is the only of the three systems to support hierarchical async-finish natively via the use of a special "finish-EDT".

[6] Incidentally this is the same approach as chosen by the OCR team when implementing CnC running on top of OCR.

CnC, SWARM and OCR run on both shared and distributed memory systems (OCR extensions in progress for distributed systems). In this work, we only target the shared memory implementations.

4.8 Runtime Support for Hierarchical Async-Finish

In this section we discuss the support for hierarchical async-finish tasks in OCR, SWARM and CnC. Our tool automatically generates EDTs that conform to such a hierarchical execution model from sequential input code.

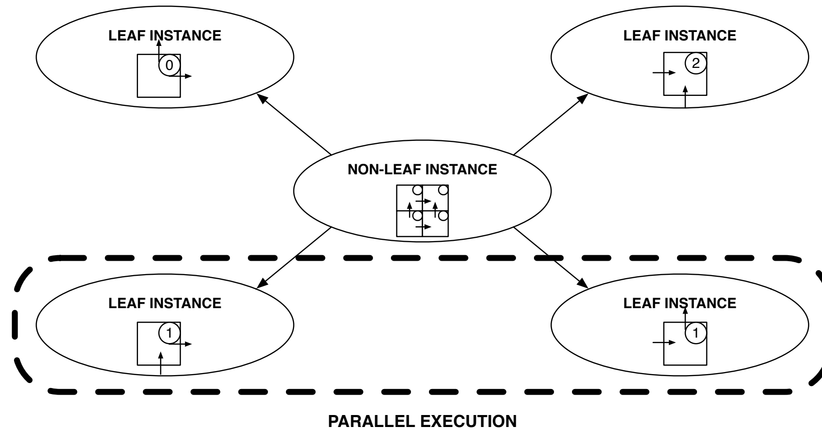

Figure 11: Parallelism among hierarchical WORKEREDTs.

Figure 11 illustrates parallelism across hierarchical WORKER EDTs. WORKER instances in the non-leaf worker (center circle) are connected by point-to-point dependences. Within each top-level WORKER, bottom-level WORKER are spawned, themselves connected by point-to-point dependences. Instances not connected by dependences (i.e. the unordered bottom-left and bottom-right instances in this small example) can be executed in parallel by the runtime. This is a coarse level of parallelism. Additionally, within each leaf worker, finer grained parallelism is also exploited by the runtime.

OCR natively supports hierarchical async-finish by virtue of the “finish EDT.” OCR also provides “latch” objects that can be used to emulate this feature like in SWARM (see below). The other two runtimes do not currently provide native support and we construct a layer of emulation that our source-to-API compiler targets automatically.

SWARM natively supports “counting dependence” objects which are similar to OCR latches . We use this feature as follows:

- within each STARTUP we generate code which determines how many WORKER are spawned. A swarm_Dep_t object is allocated and default initialized to this value,

- when both the counter and the counting dependence are ready, a SHUTDOWN is chained to await on the dependence object with the associated count value. When the dependence count reaches zero, the SHUTDOWN is awoken,
- a pointer to the swarm_Dep_t object is passed as a parameter into the tag of each WORKER instance. At this point, the current instance of STARTUP can spawn all its WORKERs,
- when multiple levels of hierarchy are involved, each instance of a leaf WORKER satisfies the dependence to the SHUTDOWN spawned by their common enclosing STARTUP,
- non-leaf WORKER relegate the dependence satisfaction to the SHUTDOWN spawned by the same STARTUP instance.

SHUTDOWN satisfies the counting dependence of their caller, up until the main SHUTDOWN which stops the runtime.

CnC does not natively support async-finish or even counting dependences. A reduction operator is under development but yet unavailable to us at the time of this writing. We settle for a simple solution using a C++11 `atomic<int>`. Each WORKER, upon completion performs an atomic decrement of the shared counter. Like for SWARM, the counter is constructed and passed by calling STARTUP. Unlike SWARM, the ability to notify the SHUTDOWN on the event that the counter reaches zero is missing. To perform this synchronization in CnC, a SHUTDOWN performs a get of an item that is only put in the corresponding item collection by the unique WORKER EDT that decrements the counter to zero (i.e. the dynamically "last" one). Unlike SWARM and OCR, which provide their own mechanisms, this emulation relies on the item collection (a hashtable) to perform the signaling. However, accesses this hashtable are very rare: only the last WORKER and the associated SHUTDOWN write and read it respectively.

5 Experiments

The numbers we present in this experimental section should be viewed as a baseline performance achievable from a sequential specification automatically translated into EDTs before single thread tuning is applied and in the absence of data and code placement hints to the runtime. In particular, **no single thread performance optimization** for SIMD, no data-layout transformation and no tile size selection heuristic or tuning are applied except where specified.[7] The mapping decisions are **the same in all EDT cases** except where specified[8]. Tile sizes for EDTs in our experiments are fixed to 64 for the innermost loops and 16 for non-innermost loops. This is by no means optimal but just a heuristic

[7] Thus there is a potential performance improvement from such transformations beyond the benefits described here.

[8] In fact, the generated files differ only in their includes and handling of the SHUTDOWN EDT.

Benchmark	EDT version	1 th.	2 th.	4 th.	8 th.	16 th.	32 th.
DIV-3D-1	DEP	2.53	4.51	6.52	6.65	**7.24**	5.27
	BLOCK	2.54	4.41	5.87	**6.91**	6.72	5.09
	ASYNC	2.82	2.82	2.62	2.60	2.36	2.18
FDTD-2D	DEP	1.19	2.21	4.06	7.19	**10.85**	10.03
	BLOCK	1.12	2.16	3.98	6.88	8.44	7.94
	ASYNC	1.17	2.23	4.31	7.94	**12.20**	10.73
GS-2D-5P	DEP	0.98	1.90	3.04	5.27	**8.45**	**10.85**
	BLOCK	0.81	1.56	2.90	4.58	4.86	4.45
	ASYNC	0.84	1.65	3.13	5.75	6.26	5.25
GS-2D-9P	DEP	0.98	1.96	3.30	5.67	9.42	**13.72**
	BLOCK	0.91	1.76	3.34	5.94	8.30	7.23
	ASYNC	0.95	1.85	3.61	6.56	**11.29**	8.96
GS-3D-27P	DEP	1.82	3.63	6.54	11.74	21.89	32.14
	BLOCK	1.78	3.61	6.93	12.87	24.78	**37.41**
	ASYNC	1.81	3.56	6.88	12.89	24.64	**37.77**
GS-3D-7P	DEP	1.60	3.19	5.02	8.46	14.80	21.56
	BLOCK	1.52	3.05	5.84	10.81	20.58	**33.06**
	ASYNC	1.54	3.06	5.87	10.93	20.40	**33.89**
JAC-2D-COPY	DEP	4.28	7.57	12.79	21.43	**28.73**	23.39
	BLOCK	3.53	6.71	11.39	14.17	13.34	13.22
	ASYNC	3.81	7.28	13.77	23.93	**26.21**	23.92
JAC-2D-5P	DEP	2.05	4.01	5.96	8.92	13.51	**16.88**
	BLOCK	1.57	2.96	5.49	8.48	8.80	8.16
	ASYNC	1.58	3.15	5.98	10.76	**16.35**	10.17
JAC-2D-9P	DEP	1.46	3.06	5.56	10.24	17.54	17.78
	BLOCK	1.51	2.86	5.47	9.73	15.09	15.76
	ASYNC	1.55	2.96	5.58	10.72	**19.13**	**18.80**
JAC-3D-27P	DEP	2.42	4.75	8.72	15.66	25.69	28.10
	BLOCK	2.41	4.90	9.20	16.77	31.64	**34.96**
	ASYNC	2.39	4.68	8.95	16.68	31.74	**35.15**

Benchmark	EDT version	1 th.	2 th.	4 th.	8 th.	16 th.	32 th.
JAC-3D-1	DEP	2.58	4.89	8.17	**9.36**	8.38	7.36
	BLOCK	3.14	4.68	7.62	**9.67**	8.71	6.53
	ASYNC	2.77	3.55	3.26	2.90	2.67	1.78
JAC-3D-7P	DEP	2.21	4.21	7.03	11.72	17.07	19.09
	BLOCK	2.28	4.54	7.78	14.50	25.16	**26.37**
	ASYNC	2.18	4.11	7.76	14.19	25.82	**27.00**
LUD	DEP	2.45	4.15	4.28	5.28	**7.47**	**7.14**
	BLOCK	1.05	1.94	2.45	2.41	1.76	1.59
	ASYNC	1.38	2.47	3.34	3.09	2.71	2.50
MATMULT	DEP	4.29	8.44	15.64	25.20	**43.30**	40.63
	BLOCK	4.24	8.51	15.27	28.34	42.02	**42.17**
	ASYNC	3.96	4.12	4.05	4.12	3.59	2.57
P-MATMULT	DEP	1.36	2.75	3.68	5.61	6.72	**8.44**
	BLOCK	1.26	2.62	4.55	7.25	7.62	6.24
	ASYNC	1.28	2.48	4.66	8.37	**9.33**	5.58
POISSON	DEP	0.65	1.00	1.54	2.13	**2.57**	**2.29**
	BLOCK	0.25	0.48	0.49	0.48	0.36	0.27
	ASYNC	0.35	0.55	0.63	0.59	0.53	0.47
RTM-3D	DEP	3.05	5.49	9.68	16.79	**22.30**	19.64
	BLOCK	3.06	5.76	9.60	17.11	**21.78**	18.73
	ASYNC	3.26	2.95	3.09	3.10	2.73	2.56
SOR	DEP	0.57	0.98	1.43	1.93	**2.46**	**2.40**
	BLOCK	0.19	0.59	0.49	0.41	0.41	0.35
	ASYNC	0.27	0.51	0.50	0.45	0.42	0.40
STRSM	DEP	5.82	10.51	14.77	20.29	**22.77**	**22.26**
	BLOCK	2.75	7.78	6.00	5.01	3.84	2.94
	ASYNC	3.50	6.23	6.82	6.39	5.69	5.07
TRISOLV	DEP	1.96	3.58	4.13	5.81	**7.90**	**8.39**
	BLOCK	1.11	2.79	2.98	2.35	1.79	1.76
	ASYNC	1.30	2.40	3.47	3.23	1.87	2.63

Table 1: CnC performance in Gflops/s

for overdecomposition to occur while keeping a reasonable streaming prefetch and single thread performance. We also compare our results to automatically generated OMP using our framework which includes a static heuristic for tile size selection [MVW+11, LLM+08]. The static tile sizes selected for OMP are meant to load-balance the execution over a statically fixed number of cores and is also mindful of streaming memory engines.

Table 2 gives a characterization of the experiments we run. For each benchmark, we specify whether it contains symbolic parameters (and if so how many), the data and iteration space size as well as the number of EDTs we generated and the maximum number of floating point operations per full EDT(at the tile size granularities described above). In order to characterize latencies and stress-test the different runtimes, the experiments we run are diverse in their sizes, running from a mere 53 *ms* in single thread sequential mode (JAC-3D-1) up to 97 *s* (JAC-3D-27P).

Experiments are performed on a two socket, eight core per socket Intel Sandy Bridge E5-2690 @ 2.90GHz running Fedora Core 19. Each core is additionally hyperthreaded for a maximum of 32 threads of execution. All experiments were run using "g++-4.8.0 -O3" and linked with our C++ RAL that we targeted to Intel's CnC v0.8, ETI's SWARM v0.13, and the Open Community Runtime (OCR) v0.8. No restriction on processors has been enforced by using numactl. We did not observe differences worth reporting with numactl on these examples.

5.1 CnC Dependence Specification Alternatives

We present CnC results separately because CnC allows three different modes of specifying dependences which already give interesting insights. The default RAL for CnC uses blocking *get* and is referred to as BLOCK. This mechanism may introduce unnecessary overhead. We also retargeted the RAL to target CnC's *unsafe_get*/*flush_gets* mechanism to provide more asynchrony. This mechanism is similar conceptually to the non-blocking gets in SWARM. A third CnC mechanism is the so-called *depends* mechanism. For each task, all its dependences are pre-specified at the time of task creation. This is similar in philosophy to the PRESCRIBEREDT that we generate automatically for OCR. Table 1 shows the baseline performance achieved by our CnC generated codes when varying the way dependences are specified. Unsurprisingly, blocking gets result in significant overheads in cases where many smaller EDTs are generated, which require more calls into the runtime. This effect is not problematic in the larger 3D cases. More surprising is the fact that DEP performs significantly worse in the cases GS-3D-7P, GS-3D-27P, JAC-3D-7Pand JAC-3D-27P. We conjecture this is not due to runtime overhead but to scheduling decisions. To confirm this, we run the following experiment: we generate 2 levels of hierarchical EDTs (which effectively increases the potential runtime overhead for DEP). In these codes, the non-leaf WORKER has the granularity of the two outermost loops, whereas the leaf WORKER has the granularity of an original EDT(16-16-16-64). Despite the increased runtime overhead to manage these nested tasks, we obtain up to 50% speedup, as shown in Table 3.

Benchmark	Type	Data size	Iteration size	# EDTs	# Fp / EDT
DIV-3D-1	Param. (1)	256^3	256^3	1 K	128 K
FDTD-2D	Const.	1000^2	$500 \cdot 1000^2$	148 K	48 K
GS-2D-5P	Param. (2)	1024^2	$256 \cdot 1024^2$	16 K	80 K
GS-2D-9P	Param. (2)	1024^2	$256 \cdot 1024^2$	16 K	144 K
GS-3D-27P	Param. (2)	256^3	256^4	256 K	6.75 M
GS-3D-7P	Param. (2)	256^3	256^4	256 K	1.75 M
JAC-2D-COPY	Const.	1000^2	1000^3	60 K	80 K
JAC-2D-5P	Param. (2)	1024^2	$256 \cdot 1024^2$	16 K	80 K
JAC-2D-9P	Param. (2)	1024^2	$256 \cdot 1024^2$	16 K	144 K
JAC-3D-27P	Param. (2)	256^3	256^4	256 K	6.75 M
JAC-3D-1	Param. (2)	256^3	256^3	1 K	112 K
JAC-3D-7P	Param. (2)	256^3	256^4	256 K	1.75M
LUD	Const.	1000^2	$1000^3/8$	60 K	10 K
MATMULT	Const.	1024^2	1024^3	64 K	32 K
P-MATMULT	Const.	256^2	$\sum_{i=1}^{256} i^3$	1 K	32 K
POISSON	Const.	1024^2	$32 \cdot 1024^2$	11 K	96 K
RTM-3D	Param. (2)	256^3	256^3	1 K	512 K
SOR	Const.	$10,000^2$	$10,000^2$	10 M	5 K
STRSM	Const.	1500^2	1500^3	200 K	16 K
TRISOLV	Const.	1000^2	1000^3	60 K	16 K

Table 2: Benchmark characteristics

5.2 SWARM, OCR and OpenMP

We now turn to numbers we obtain with SWARM, OCR and OpenMP. We break down the discussion in different categories of benchmarks. The discussion here is also valid for CnC.

1) Embarrassingly Parallel Examples are ones for which no runtime dependences are required (DIV-3D-1, JAC-3D-1 RTM-3D and MATMULT). The execution times for the first 3 examples are very low ($53 - 210$ ms on 1 thread), and can be viewed as a test of runtime latency overhead on very short runs, without dependences. MATMULT is a somewhat larger example. These examples show SWARM has a smaller overhead than CnC and OCR for running parallel tasks, until reaching the hyperthreading mode where SWARM performance consistently drops. One could argue whether hyperthreading should be accounted for; since it shows interesting and different behaviors across runtimes, we chose to report it.

2) EDT granularity. LUD, POISSON and SOR illustrate quite small examples in which the statically selected tile sizes are not adequate for EDT granularity purposes. In the case of POISSON pipeline startup cost is prohibitively expensive; choosing tile sizes of $2 - 32 - 128$ yields around 7 Gflop/s with OCR on 32 threads, a 6x speedup. In the case of SOR, the tile sizes yield small tasks of merely 1024 iterations corresponding to 5K instructions; selecting larger tile sizes also improves performance. Overall, the examples in this section show that relatively small tile sizes that achieve overprovisioning may not be beneficial. We discuss this further for SOR and LUD in the next section.

3) OpenMP Efficient Examples. STRSM and TRISOLV illustrate 2 cases that mix both parallel and permutable loops and for which OpenMP performs significantly better than any of the EDT solutions. In this case, we could narrow

Benchmark	version	1 th.	2 th.	4 th.	8 th.	16 th.	32 th.
GS-3D-7P	DEP	1.61	3.28	6.21	11.46	21.20	32.88
GS-3D-27P	DEP	1.82	3.62	6.94	13.03	25.02	34.83
JAC-3D-7P	DEP	2.12	3.91	7.44	13.19	20.18	25.11
JAC-3D-27P	DEP	2.38	4.81	8.79	16.04	30.78	33.31

Table 3: CnC, two level hierarchy, performance in Gflops/s

the problem down to tile size selection for reuse. In the case of STRSM, by selecting square tiles of size $64 - 64 - 64$, we were able to obtain up to 76 Gflop/s with OCR. Unfortunately the performance would not increase further with hyperthreading. In addition, forcing the OpenMP tile sizes to $16 - 16 - 64$ capped the performance at 50 Gflop/s. In the case of TRISOLV, by selecting a tile of size $64 - 64 - 256$,we were able to obtain up to 26 Gflop/s with OCR. This further emphasizes the need for a proper tile size selection in EDT-based runtimes. There is a difficult trade-off between over-decomposition, reuse, single thread performance, streaming prefetch utilization and problem size that should be solved in a dynamic and adaptive fashion. To allow this adaptivity in the future, we will generate parametrized tasks that can recursively decompose and expose the proper parameters to the runtime.
4) 2-D and 3-D Time Tiling. The remaining examples show the benefit of EDTs. As expected in those cases, performance for EDTbased codes scales significantly better than OpenMP performance, especially as the Jacobi examples (explicit relaxation scheme) move twice as much memory as GaussSeidel examples (implicit relaxation scheme) and do not scale as well from 16 to 32 threads (hyperthreading). Still in the future they are expected to scale better with more processors because they can be subjected to diamond tiling[BPB12] that presents a parallel start-up front and significantly more parallelism than time tiling schemes. At 3-D levels of granularity, we see no significant difference between the 3 EDT-based runtimes we targeted.

5.3 Effects of EDT Granularity

We further investigated the examples of LUD and SOR on which EDT performance was lower than we expected. We investigated a few different tile sizes as well as 2 levels of granularity (for LUD). The granularity parameter represents the number of loops in an EDT. Figure 5 shows there is a fine trade-off between EDT granularity, number of EDTs and cost of managing these EDTs. To confirm the runtime overhead as EDTs shrink in size, we additionally collected performance hotspots using Intel Vtune amplxe-cl for LUD16-16-16 with granularity 3 and 4 at 16 threads. First, we did not observe templated expressions calculations appearing, confirming the low extra overhead of our solution. Second, in the case of granularity 4, more than 85% of the non-idle time is spent executing work, the rest being spent mostly in the OCR dequeInit function. However, at the finer granularity, the ratio of effective work drops to merely 10% stealing and queue management taking up to 80%. The drop in performance between

Benchmark	EDT version	1 th.	2 th.	4 th.	8 th.	16 th.	32 th.
DIV-3D-1	OCR	2.44	3.89	4.84	6.64	6.43	5.63
	OMP	3.02	5.65	7.62	**8.86**	**8.76**	8.46
	SWARM	1.91	4.00	6.38	8.06	8.28	2.96
FDTD-2D	OCR	1.20	2.31	4.34	8.13	**13.96**	**17.14**
	OMP	0.83	0.29	0.56	0.95	1.41	2.46
	SWARM	1.17	2.07	3.91	7.46	11.91	13.75
GS-2D-5P	OCR	0.89	1.72	3.23	5.90	**10.38**	**15.04**
	OMP	1.13	1.14	1.16	1.19	1.22	1.28
	SWARM	0.88	1.65	3.11	5.74	9.73	3.11
GS-2D-9P	OCR	0.98	1.90	3.61	6.67	**12.05**	**18.24**
	OMP	1.17	1.16	1.18	1.17	1.19	1.20
	SWARM	0.96	1.85	3.50	6.51	11.51	11.88
GS-3D-7P	OCR	1.55	3.04	5.87	10.91	20.72	**34.25**
	OMP	1.75	2.21	3.01	4.90	7.83	11.12
	SWARM	1.56	2.93	5.64	10.64	20.29	**32.71**
GS-3D-27P	OCR	1.82	3.56	6.90	12.95	24.71	**37.53**
	OMP	2.06	3.16	5.51	10.16	18.86	29.26
	SWARM	1.84	3.52	6.80	12.78	24.45	**37.19**
JAC-2D-COPY	OCR	4.05	7.57	14.34	25.66	**44.81**	**42.90**
	OMP	4.25	5.30	7.33	12.60	19.90	18.00
	SWARM	3.67	6.12	11.34	21.40	35.51	9.37
JAC-2D-5P	OCR	1.71	3.22	6.11	11.08	**18.98**	**21.72**
	OMP	0.92	0.92	0.91	1.13	1.40	2.19
	SWARM	1.63	3.15	5.84	10.63	17.58	6.15
JAC-2D-9P	OCR	1.58	3.00	5.84	10.52	18.99	**21.54**
	OMP	1.09	1.14	1.20	1.31	1.67	2.64
	SWARM	1.50	3.00	5.58	10.27	18.53	**19.48**
JAC-3D-27P	OCR	2.41	4.70	8.94	16.72	31.66	**34.48**
	OMP	2.43	3.43	5.66	10.36	18.87	25.95
	SWARM	2.40	4.53	8.75	16.21	30.67	**34.51**

Benchmark	EDT version	1 th.	2 th.	4 th.	8 th.	16 th.	32 th.
JAC-3D-7P	OCR	2.18	4.14	7.80	14.17	25.50	**26.75**
	OMP	1.93	2.15	2.62	4.12	7.42	12.66
	SWARM	2.12	3.81	7.32	13.74	24.84	**26.09**
JAC-3D-1	OCR	2.97	4.71	8.38	9.46	8.35	6.71
	OMP	3.33	5.70	11.61	**19.59**	**17.53**	13.67
	SWARM	2.16	5.91	7.93	11.14	12.14	3.18
LUD	OCR	1.66	2.72	5.21	7.33	**7.67**	4.91
	OMP	0.57	0.78	0.94	0.67	0.59	0.98
	SWARM	2.02	2.93	4.70	6.91	**7.71**	1.35
MATMULT	OCR	4.37	8.35	15.05	26.80	**45.72**	43.77
	OMP	1.21	2.38	4.49	8.37	15.78	14.41
	SWARM	4.46	8.58	15.49	28.87	**49.44**	35.53
P-MATMULT	OCR	1.37	2.59	4.89	8.81	14.46	15.60
	OMP	1.90	2.97	5.48	9.12	**15.98**	**20.14**
	SWARM	1.35	2.66	5.05	9.35	13.55	3.82
POISSON	OCR	0.46	0.64	1.14	**1.71**	1.43	1.00
	OMP	1.01	0.97	1.02	0.99	0.96	0.84
	SWARM	0.44	0.63	0.99	1.41	**1.57**	0.27
RTM-3D	OCR	3.00	5.38	9.65	15.84	24.42	17.48
	OMP	2.40	4.59	8.03	15.77	**29.06**	22.67
	SWARM	2.83	5.95	9.76	18.02	**26.23**	12.34
SOR	OCR	0.28	0.56	0.98	1.65	1.27	0.93
	OMP	0.62	1.01	1.59	2.66	**4.42**	**6.62**
	SWARM	0.26	0.45	0.68	1.17	0.86	0.22
STRSM	OCR	4.49	7.58	11.76	17.62	15.95	11.72
	OMP	3.66	5.60	10.52	19.84	**37.97**	**39.15**
	SWARM	2.82	4.15	7.39	13.04	17.88	2.79
TRISOLV	OCR	1.64	2.95	4.89	7.63	7.55	5.29
	OMP	2.09	4.29	7.77	15.15	**28.67**	**23.28**
	SWARM	1.56	2.84	4.88	7.88	9.77	1.37

Table 4: SWARM, OCR and OpenMP performance in Gflops/s

Benchmark	Sizes	Gran.	1 th.	2 th.	4 th.	8 th.	16 th.	32 th.
LUD	16-16-16	3	0.92	1.54	2.55	3.49	2.04	1.41
LUD	16-16-16	4	1.85	3.66	6.63	11.44	**16.68**	9.56
LUD	64-64-64	3	2.41	4.66	8.01	13.34	14.15	10.80
LUD	64-64-64	4	1.94	3.63	5.73	6.54	6.22	3.46
LUD	10-10-100	3	1.90	3.57	6.55	11.64	**16.25**	10.67
LUD	10-10-100	4	1.83	3.56	6.63	11.64	14.39	9.93
SOR	100-100	2	0.64	1.21	2.28	4.15	6.85	5.18
SOR	100-1000	2	0.64	1.28	2.09	3.46	4.84	4.18
SOR	200-200	2	0.63	1.26	2.31	4.28	**7.04**	**8.16**
SOR	1000-1000	2	0.67	1.21	2.03	2.79	2.97	2.73

Table 5: OCR, tile size exploration, performance in Gflops/s

16-16-16 and 10-10-100 suggests there is a critical threshold, possibly linked to last-level cache sizes, at which the overhead of OCR increases substantially. While these initial performance results are encouraging when the granularity selection is adequate, the overhead introduced to manage fine-grained tasks is too high at the moment to allow the envisioned level of overdecomposition. Still the OCR implementation is in the early stages and increasing performance is to be expected.

6 Related Work

The most closely related work to ours is [BVB+09], but our contribution presents significant differences with this work.

First, our scheme is generic, targets three different runtimes and is easily extensible to other runtimes based on task graphs like OCR or on tag-tuples like CnC and SWARM. The experiments section shows the variability between runtimes and the benefit of a nimble strategy. Second, our scheme is decentralized and fully asynchronous in the creation of tasks and the dependences between them. Baskaran's solution must first construct the full graph and then only begin useful work. Considering Amdahl's law, our solution is expected to scale better on larger numbers of processors and distributed memory. Third, our baseline dependence specification mechanism is scalable at both compile-time and runtime by virtue of exploiting loop types and dependence information on restructured loops available from the scheduler. Last, loop type information is a general compiler concept and has a chance to be extended to more general dependence analyses [OR12]. Our expression template dependence resolution and hierarchical decomposition mechanism are first steps in this direction.

There are a number of other runtimes that our mapping algorithms could target. Most successfully, the QUARK runtime [HLYD11, YKD11], speeds the PLASMA linear algebra library [DL11] with dynamic task scheduling and a task-oriented execution model. However, PLASMA is parallelized to QUARK *by hand*. Our mapping approach may be used to more rapidly and portably *automatically* generate task-oriented implementations of linear algebra to QUARK.

Then, it could be used to regenerate implementations of such a linear algebra library taking advantage of the features of CnC, SWARM, and OCR. Furthermore, our approach is directly oriented toward porting the library for impending architectural changes from exascale, such as very deep memory hierarchies. Other EDT oriented runtimes that our mapping approach could target include The Qthreads Library [WMT08] or HPX [TAB+11].

7 Conclusions

We have demonstrated the first fully automatic solution for generating event-driven, tuple-space based programs from a sequential C specification for multiple EDT-based runtimes. Our solution performs hierarchical mapping and exploits hierarchical async-finishes. Our solution uses auto-parallelizing compiler technology to target three different runtimes relying on event-driven tasks (EDTs) via a runtime-agnostic C++ layer, which we have retargeted to Intel's Concurrent Collections (CnC), ETI's SWARM, and the Open Community Runtime (OCR). We have further demonstrated the performance improvements achievable with these EDT-based runtimes and pinpointed some current weaknesses related to fine-grained EDT management overhead. Our solution takes advantage of parallel and permutable loops to abstract aggregate dependences between EDTs. The notion of permutability is a general compiler concept and the transformations we propose can be extended to more general programs than we currently handle. Lastly, our templated expression based multi-dimensional spaces can be extended to decompose recursively and adapt at runtime, which will be the subject of future work.

References

[Bas04a] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'04*, pages 7–16, Juan-les-Pins, September 2004.

[Bas04b] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *IEEE PACT*, pages 7–16. IEEE Computer Society, 2004.

[BBC+10] Zoran Budimlic, Michael G. Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David M. Peixotto, Vivek Sarkar, Frank Schlimbach, and Sagnak Tasirlar. Concurrent collections. *Scientific Programming*, 18(3-4):203–217, 2010.

[BHRS08] U. Bondhugula, A. Hartono, J. Ramanujan, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM SIGPLAN Programming Languages Design and Implementation (PLDI '08)*, Tucson, Arizona, June 2008.

[BHT+10] Muthu Manikandan Baskaran, Albert Hartono, Sanket Tavarageri, Thomas Henretty, J. Ramanujam, and P. Sadayappan. Parameterized tiling revisited. In Andreas Moshovos, J. Gregory Steffan, Kim M. Hazelwood, and David R. Kaeli, editors, *CGO*, pages 200–209. ACM, 2010.

[BJK+96] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *J. Parallel Distrib. Comput.*, 37(1):55–69, 1996.

[Boa08] OpenMP Architecture Review Board. The openmp specification for parallel programming `http://www.openmp.org`, 2008.

[BPB12] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. Tiling stencil computations to maximize parallelism. In *SC*, page 40, 2012.

[BVB+09] Muthu Manikandan Baskaran, Nagavijayalakshmi Vydyanathan, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. In Daniel A. Reed and Vivek Sarkar, editors, *PPOPP*, pages 219–228. ACM, 2009.

[CA88] David E. Culler and Arvind. Resource requirements of dataflow programs. In Howard Jay Siegel, editor, *ISCA*, pages 141–150. IEEE Computer Society, 1988.

[Car13] Nicholas Carter. et al. runnemede: An architecture for ubiquitous high-performance computing. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, pages 198–209, Washington, DC, USA, 2013. IEEE Computer Society.

[CBF95] J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 92–101, Santa Barbara, California, July 1995.

[CI95] Béatrice Creusillet and Francois Irigoin. Interprocedural Array Region Analyses. In *Eighth International Workshop on Languages and Compilers for Parallel Computing (LCPC'95)*, pages 4–1 to 4–15, August 1995.

[Den74] Jack B. Dennis. First version of a data flow procedure language. In Bernard Robinet, editor, *Symposium on Programming*, volume 19 of *Lecture Notes in Computer Science*, pages 362–376. Springer, 1974.

[DL11] Jack Dongarra and Piotr Luszczek. Plasma. In Padua [Pad11], pages 1568–1570.

[Fea88] P. Feautrier. Parametric integer programming. *RAIRO-Recherche Opérationnelle*, 22(3):243–268, 1988.

[Fea91] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–52, February 1991.

[Fea92] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, October 1992.

[FLPR99] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *FOCS*, pages 285–298. IEEE Computer Society, 1999.

[GFL99] M. Griebl, P. Feautrier, and C. Lengauer. Index set splitting. *International Journal of Parallel Programming*, 28:607–631, July 1999.

[Gri04] Martin Griebl. Automatic parallelization of loop programs for distributed memory architectures, 2004. Habilitation thesis.

[GVB+06] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Int. J. Parallel Program.*, 34(3):261–317, June 2006.

[HBB+09] A. Hartono, M. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorthy, B. Norris, J. Ramanujam, and P. Sadayappan. Parametric multi-level tiling of imperfectly nested loops. In *Proceedings of the ACM International Conference on Supercomputing (ICS'09)*, pages 147–157, Yorktown Heights, New York, June 2009.

[HLYD11] Azzam Haidar, Hatem Ltaief, Asim YarKhan, and Jack Dongarra. Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures. *Concurrency and Computation: Practice and Experience*, 24(3):305–321, 2011.

[IHBZ10] Costin Iancu, Steven A. Hofmeyr, Filip Blagojevic, and Yili Zheng. Oversubscription on multicore processors. In *IPDPS*, pages 1–11. IEEE, 2010.

[Inta] Intel. Concurrent collections. http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/.

[Intb] ET International. SWift adaptive runtime machine. http://www.etinternational.com/index.php/products/swarmbeta/.

[IT88] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, pages 319–329, New York, NY, USA, January 1988. ACM Press.

[JCP+12] Alexandra Jimborean, Philippe Clauss, Benoît Pradelle, Luis Mastrangelo, and Vincent Loechner. Adapting the polyhedral model as a framework for efficient speculative parallelization. In *PPOPP*, pages 295–296, 2012.

[KAH+12] Himanshu Kaul, Mark Anders, Steven Hsu, Amit Agarwal, Ram Krishnamurthy, and Shekhar Borkar. Near-threshold voltage (ntv) design: opportunities and challenges. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *DAC*, pages 1153–1158. ACM, 2012.

[KBB+07] Sriram Krishnamoorthy, Muthu Manikandan Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *PLDI*, pages 235–244, 2007.

[KBC+08] P. M. Kogge, Shekhar Borkar, William W. Carlson, William J. Dally, Monty Denneau, Paul D. Franzon, Stephen W. Keckler, Dean Klein, Robert F. Lucas, Steve Scott, Allan E. Snavely, Thomas L. Sterling, R. Stanley Williams, Katherine A. Yelick, William Harrod, Daniel P. Campbell, Kerry L. Hill, Jon C. Hiller, Sherman Karp, Mark A. R.s, and Alfred J. Scarpelli. Exascale Study Group: Technology Challenges in Achieving Exascale Systems. Technical report, DARPA, 2008.

[LLM+08] R. Lethin, A. Leung, B. Meister, P. Szilagyi, N. Vasilache, and D. Wohlford. Final report on the R-Stream 3.0 compiler DARPA/AFRL Contract # F03602-03-C-0033, DTIC AFRL-RI-RS-TR-2008-160. Technical report, Reservoir Labs, Inc., May 2008.

[LRW91] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In David A. Patterson, editor, *ASPLOS*, pages 63–74. ACM Press, 1991.

[LVML09] Allen K. Leung, Nicolas T. Vasilache, Benoit Meister, and Richard A. Lethin. Methods and apparatus for joint parallelism and locality optimization in source code compilation, September 2009.

[MS68] T. H. Myer and Ivan E. Sutherland. On the design of display processors. *Commun. ACM*, 11(6):410–414, 1968.

[MVW+11] Benoît Meister, Nicolas Vasilache, David Wohlford, Muthu Manikandan Baskaran, Allen Leung, and Richard Lethin. R-stream compiler. In Padua [Pad11], pages 1756–1765.

[OR12] Cosmin E. Oancea and Lawrence Rauchwerger. Logical inference techniques for loop parallelization. In *Proceedings of the 33rd ACM*

SIGPLAN conference on Programming Language Design and Implementation, PLDI '12, pages 509–520, New York, NY, USA, 2012. ACM.

[Pad11] David A. Padua, editor. *Encyclopedia of Parallel Computing.* Springer, 2011.

[PSP98] Department of Mathematics P. S. Pacheco. *A User's Guide to MPI*, March 1998.

[PW92] William Pugh and David Wonnacott. Eliminating false data dependences using the omega test. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, pages 140–151, New York, NY, USA, 1992. ACM.

[RAO98] Lawrence Rauchwerger, Francisco Arzu, and Koji Ouchi. Standard templates adaptive parallel library (stapl). In *LCR*, pages 402–409, 1998.

[Rei07] James Reinders. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism.* O'Reilly, 2007.

[RKRS07] Lakshminarayanan Renganarayanan, DaeGon Kim, Sanjay V. Rajopadhye, and Michelle Mills Strout. Parameterized tiled loops for free. In *PLDI*, pages 405–414, 2007.

[SGS+93] Ellen Spertus, Seth Copen Goldstein, Klaus E. Schauser, Thorsten von Eicken, David E. Culler, and William J. Dally. Evaluation of mechanisms for fine-grained parallel programs in the j-machine and the cm-5. In Alan Jay Smith, editor, *ISCA*, pages 302–313. ACM, 1993.

[TAB+11] Alexandre Tabbal, Matthew Anderson, Maciej Brodowicz, Hartmut Kaiser, and Thomas L. Sterling. Preliminary design examination of the parallex system from a software and hardware perspective. *SIGMETRICS Performance Evaluation Review*, 38(4):81–87, 2011.

[TCK+11] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The Pochoir Stencil Compiler. In *SPAA '11: The 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, San Jose, California, June 2011.

[tea] The OCR team. The ocr project. https://01.org/projects/open-community-runtime.

[WMT08] Kyle B. Wheeler, Richard C. Murphy, and Douglas Thain. Qthreads: An API for programming with millions of lightweight threads. In *IPDPS*, pages 1–8. IEEE, 2008.

[YAK00] Qing Yi, Vikram S. Adve, and Ken Kennedy. Transforming loops to recursion for multi-level memory hierarchies. In Monica S. Lam, editor, *PLDI*, pages 169–181. ACM, 2000.

[YKD11] A. YarKhan, J. Kurzak, and J. Dongarra. Quark users' guide: Queueing and runtime for kernels. Technical Report ICL-UT-11-02, University of Tennessee Innovative Computing Laboratory, 2011.