Mapping Stream Programs onto Heterogeneous Multiprocessor Systems

Paul M. Carpenter
paul.carpenter@bsc.es

Alex Ramirez
alex.ramirez@bsc.es

Eduard Ayguade
eduard.ayguade@bsc.es

Barcelona Supercomputing Center
c/ Jordi Girona, 31
08034 Barcelona, Spain

ABSTRACT

This paper presents a partitioning and allocation algorithm for an iterative stream compiler, targeting heterogeneous multiprocessors with constrained distributed memory and any communications topology. We introduce a novel definition of connectedness that enables the algorithm to model the capabilities of the compiler. The algorithm uses convexity and connectedness constraints to produce partitions that are easier to compile and require short pipelines. Software pipelining is an effective transformation, but it increases memory footprint and latency, and has a startup overhead. Our algorithm takes account of these downstream costs.

We show results for the StreamIt 2.1.1 benchmarks for an SMP, 2×2 mesh, SMP plus accelerator, and IBM QS20 blade, which has two Cell processors. Our results show that the average performance is within 5% of the unrestricted optimum found using a brute force search, while seldom requiring software pipelining. The heuristic is robust, and fast enough to be inside the feedback loop of an iterative compiler.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms

Algorithms, Performance

Keywords

Stream Programming, Partitioning, Convexity, Multicore

1. INTRODUCTION

Recent trends in computer architecture point to an increasing number of on-chip processor cores, particularly in the portable embedded space [26]. Many embedded systems contain multiple ISAs or microarchitectures, both to optimize algorithm-specific cores and to mitigate Amdahl's law [21]; e.g. Intel IXP2850 [16], TI OMAP [7], Nexperia Home Platform [9], and ST Nomadik [2]. Programs using explicit threads have to be tuned to match the target, and often require updating to use increasing numbers of cores. Hence the interest in automatically mapping a portable program onto a heterogeneous multiprocessor.

Many multimedia and radio applications contain abundant task and data parallelism, but it is hard to extract from C code. Stream languages represent the application as independent actors communicating via point-to-point streams, which is natural for signal processing, is deterministic, and exposes concurrency to the compiler.

This paper presents an algorithm that maps a stream program onto a heterogeneous target, fusing kernels and assigning them to processors. Although the primary goal is to achieve high resource utilisation, the choice of partition affects performance indirectly through its effect on buffer allocation and software pipelining (see Section 3.1). Since the search space is large, it is impractical to model precise memory, latency, and startup overheads of software pipelining. We use the heuristic that these costs increase with the number of pipeline stages.

We use a static partition for two reasons. Firstly, to enable compile-time optimizations between kernels in the same task; e.g. polyhedral loop transformations [8]. Secondly, task migration on a distributed memory processor requires DMA transfer to/from the local store, which must be done all at once, rather than on demand through caches. For example, a context switch on the Cell SPE requires about 30μs [14]. For the target applications, variation in complexity happens at the small timescale, rather than being divided into phases in the same way as, say, a C compiler. For independent SPEC workloads, a dynamic policy is preferable [4].

The input to the partitioning algorithm is the program graph of kernels and streams. The output is the mapping to fuse kernels to tasks, and to allocate tasks to processors. We distinguish *kernels*, which are present in the source program, from *tasks*, which are present in the executable, contain multiple kernels and are scheduled on a known processor. The partitioning problem, defined in section 4, is $\mathcal{NP}$-hard, so we approach its solution using heuristics.

The partitioning algorithm supports unstructured graphs with variable data rates and computation times. There are many streaming models of computation [27], but it is sufficient to view the kernels as Kahn processes [18], which are independent threads communicating through blocking operations on point-to-point streams. The algorithm requires

CASES'09, October 11–16, 2009, Grenoble, France.

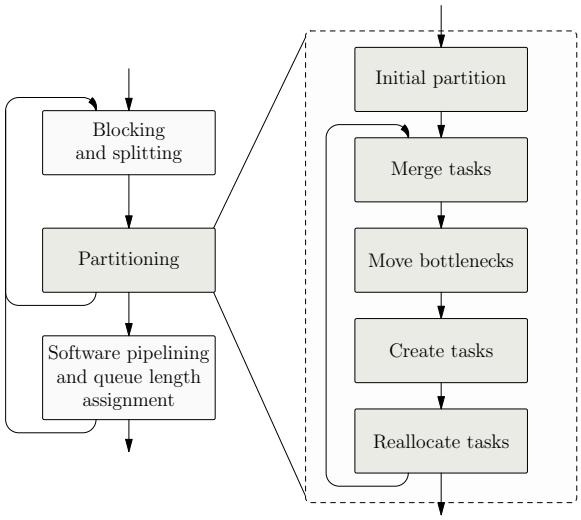


Figure 1: The mapping phase of the ACOTES compiler, showing the partitioning algorithm that is the focus of this paper (in the dashed block)

the average data rate on each stream, and the average load of each kernel on each processor. These values could be obtained from a profile or, in cases like Synchronous Data Flow [22], derived statically.

On average, across the StreamIt benchmarks, the partitions generated by the algorithm are within 5% of the performance of the optimum unrestricted partition, and in most cases do not require software pipelining.

The main contributions of this paper are:

- A partitioning approach that favours convex partitions, defined below, producing close to optimal throughput using a short software pipeline
- A new definition of connectedness that enables the algorithm to generate only partitions that the compiler can support
- An algorithm that maps unstructured stream programs onto heterogeneous systems, and is fast enough for an iterative compiler

2. THE ACOTES STREAM COMPILER

This work is part of the ACOTES European project [17], which is developing an open source stream compiler for embedded systems. This compiler will automatically map a stream program, written using the Stream Programming Model (SPM) [1], an annotated version of the C programming language, onto a multicore system, applying task fusion and blocking transformations. The target system is represented using the Abstract Streaming Machine (ASM) [6], which supports heterogeneous and homogeneous multiprocessors, with distributed or shared memory.

Figure 1 shows how the partitioning algorithm fits into this stream compiler. First, a polyhedral optimisation pass unrolls to aggregate computation and communication, and splits stateless kernels for greater parallelism. This is followed by the partitioning algorithm described in this paper, and outlined in Section 1. The final pass performs software pipelining and allocates memory for the stream buffers.

Interactions between the three phases cause a phase ordering problem, which we solve by (a) making passes aware of downstream passes, and (b) using an iterative compiler. The first is achieved using the convexity constraint, which controls the length of the pipeline, reducing the demand for stream buffers. The latter is addressed by ensuring that the partitioning algorithm is fast enough to be executed several times, if necessary, with different blocking factors.

3. CONVEX CONNECTED PARTITIONS

3.1 Convexity

A partition is convex if the graph of dependencies between tasks is acyclic. Equivalently, every directed path between two kernels in the same task is internal to that task. The convexity constraint is intended to avoid long software pipelines. A partitioning algorithm unaware of the cost of pipelining may require long pipelines for a small increase in throughput. The optimal unrestricted partition for the StreamIt 2.1.1 *serpent* benchmark [11] on two Cells is 10% faster than the optimal convex partition, but it requires 209 pipeline stages rather than 31. We did not obtain the CPLEX Solver to evaluate StreamRoller, but since it uses ILP to solve a similar problem, its result should be similar. This translates into higher memory use, which may simply not fit, as well as startup overhead and latency. Table 1 shows that partitions from our algorithm seldom require pipelining, and performance is, on average, within 5% of optimum.

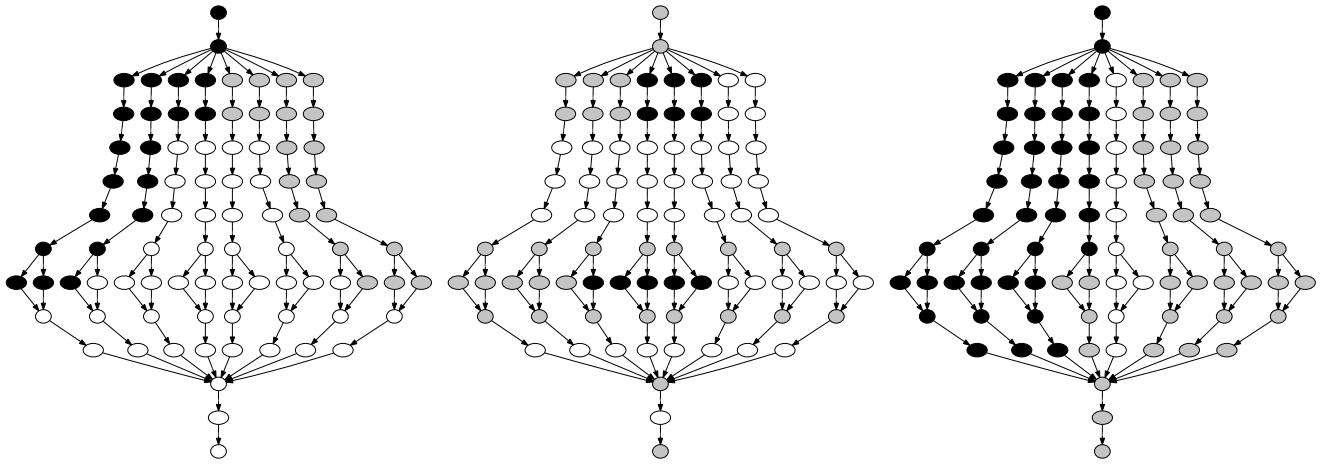
Figure 2(a) shows the convex partition from our heuristic for StreamIt *filterbank* on a 3-core SMP. Each processor has a single task containing the kernels of its colour. Data flow is from processor p_1 (black) to p_2 (grey) and p_3 (white), and from p_2 to p_3—an acyclic graph. Figure 3(a) shows an execution trace, with shades of grey corresponding to five iterations.

Figure 2(b) shows an optimal unrestricted partition. It is not convex because there is communication in both directions between p_2 (grey) and p_3 (white), so the partition is optimal only assuming perfect dynamic scheduling or software pipelining. Figure 3(b) shows a trace without software pipelining, and it has stalls where dependencies prevent computation from being overlapped, so throughput is 53% lower than the convex partition. Figure 3(c) is pipelined using the stage assignment phase from the SGMS algorithm [20]. It has 0.2% higher throughput than the convex partition, but due to startup overhead would break even only after 8,000 iterations.

When the benefit from software pipelining is above some threshold, the algorithm relaxes connectedness and convexity. Section 6 shows the partition of *vocoder*, which benefits from software pipelining. The result is close to optimal performance using a short pipeline.

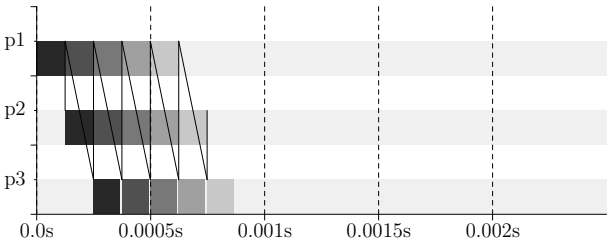
3.2 Connectivity

The connectedness constraint is primarily to help code generation, since it is easier to fuse adjacent kernels, whose relative frequencies are known via the stream between them. Figure 4(a) shows a program using SPM [1], our annotated version of C for stream programming. Kernels *read* and *write* perform IO, and *update* manages the automaton and sends only accepting states. The macros `NEW_STATE` and `ACCEPT_STATE` manage the automaton, and are irrelevant to

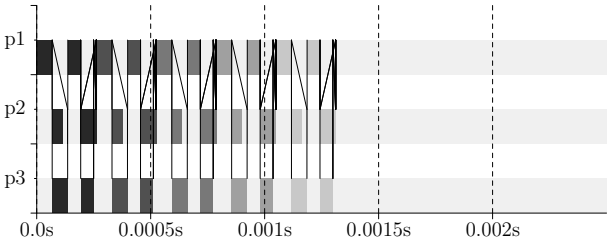


(a) Convex, loosely connected partition *(b) Unrestricted partition* *(c) Convex, strictly connected*

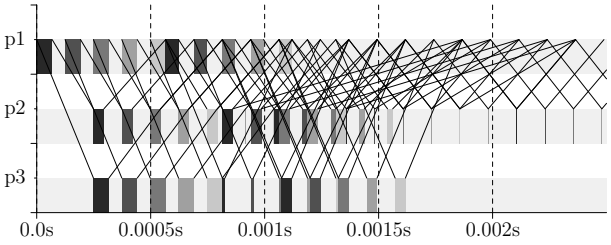
Figure 2: Example partitions of the StreamIt filterbank benchmark onto a 3-core SMP. Each node is a kernel, and each edge is a stream



(a) Convex partition from heuristic (time: 1.00)



(b) Unrestricted without pipelining (time: 2.11)



(c) Unrestricted partition with pipelining (time: 1.00)

Figure 3: Traces for five iteration of filterbank, scheduled using SGMS. Different iterations are identified using shades of grey.

the discussion. Consider the case where the partition merges *read* and *write* into task 1, with *update* in task 2. This partition is not convex, so pipelining is required. Task 1 is not connected, so the compiler requires the relative frequencies of *read* and *write*.

Both problems can be solved using dynamic scheduling, by switching between tasks when a push or a pop starts to wait. Dynamic scheduling adds overhead and unpredictibility, which are undesirable in real-time systems. In the absence of a runtime dynamic scheduler, this example requires either an extra stream carrying the condition, as in Figure 4(b), or duplicating the calculation of the condition, plus the state on which it is based, which would duplicate the whole *update* kernel.

The general case requires duplicating state or creating a dependence cycle. In Figure 5, fusing k_2 and k_3 requires fusing the entire graph. The relationship between the firing rates of k_2 and k_3 depends on all conditions on the path between them. Adding a stream from k_4 to fused k_2 and k_3, carrying some function of e and f creates a directed cycle, since there is already a stream in the other direction; that is, from k_3 to k_4.

A naïve definition of connectivity, *strict connectivity*, considers a partition to be connected when each processor has a weakly connected subgraph. Unfortunately, wide split-joins, as in *filterbank*, do not usually have good partitions subject to this constraint. In Figure 2(a), p_2 (grey) is not strictly connected, so our strict heuristic produces the partition in sub-figure (c), which has performance 28% worse than (a). In general, strict connectedness allows only the processors containing the split or the join kernel to have kernels from more than one branch.

We generalize connectivity by providing to the partitioning algorithm a set of *basic connected sets* [25], each of which specifies kernels that the compiler can pairwise merge. For strict connectivity, there is a basic connected set for each pair of communicating kernels.

This allows the partitioning algorithm to be adapted to the compiler and source language(s). If the compiler understands StreamIt [28] splitters and joiners, there should

```
int state = 0;

#pragma taskgroup
while (1)
{
   #pragma task output(c)
   int c = fgetc(fin);

   #pragma task input(c)
   {
      state = NEW_STATE(state, c);
      if (ACCEPT_STATE(state))
      {
         #pragma task input(state)
         fwrite(&state, sizeof(int), 1, fout);
      }
   }
}
```

Read: `int c = fgetc(fin);`

Update: `state = NEW_STATE(state, c);` `if (ACCEPT_STATE(state))`

Write: `fwrite(&state, sizeof(int), 1, fout);`

(a) Source code for automaton

```
void task_1(void) {
   while (1) {
      int c = fgetc(fin);
      push(stream1, c);
      if (pop(stream2)) {
         state = pop(stream3);
         fwrite(&state, sizeof(int), 1, fout);
      }
   }
}

void task_2(void) {
   int state = 0;
   while(1) {
      state = NEW_STATE(state, pop(stream1));
      cond = ACCEPT_STATE(state);
      push(stream2, cond);
      if (cond)
         push(stream3, state);
   }
}
```

Read: `int c = fgetc(fin);` `push(stream1, c);` `if (pop(stream2)) {`

Write: `state = pop(stream3);` `fwrite(&state, sizeof(int), 1, fout);`

Update: `state = NEW_STATE(state, pop(stream1));` `cond = ACCEPT_STATE(state);` `push(stream2, cond);` `if (cond)` `push(stream3, state);`

(b) Simplified compiled code

Figure 4: Motivation of connectivity: example programs with data dependent pushes and pops

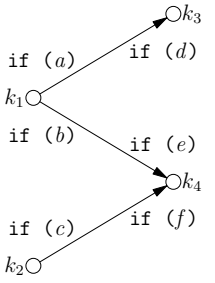


Figure 5: If k_2 and k_3 are fused into one task, then the entire graph must be fused

be a basic connected set for each splitter (joiner) containing its successors (predecessors), which solves the problem outlined above. Similarly, there may also be a basic connected set covering each region of the program graph that is internally SDF.

4. FORMALISATION OF THE PROBLEM

The target is represented as an undirected bipartite graph $H = (V, E)$, where $V = P \cup I$ is the set of vertices, a disjoint union of processors, P, and interconnects, I; and E is the set of edges. Each processor, p, has weight, w^p, equal to its clock speed in GHz, and each interconnect, u, has weight, w^u, equal to its bandwidth in GB/s. The static route between processors p and q is represented by $r^u_{pq} = 1$ if it uses interconnect u, and 0 otherwise. In general, $r^u_{pq} \neq r^u_{qp}$; e.g. dimension-order routing on a mesh. Figure 6 shows the topology of our example targets, omitting the edge and vertex weights and the routing table. This representation is a simplified form of the Abstract Streaming Machine (ASM) [6]. Figure 6(c) has two "processors", a and b, present only to achieve the correct topology, but unable to execute code; therefore $w^a = w^b = 0$.

The program is represented as a directed acyclic graph, $G = (K, S)$, where K is the set of kernels, and S is the set of streams. If the program is cyclic, then each strongly connected component is contracted into a single vertex. The load of kernel i on processor p, denoted c^{ip}, is the mean number of gigacycles in some fixed time period τ. Similarly, the load of stream ij, denoted c^{ij} is the mean number of gigabytes transferred in time τ.

The basic connected sets are a collection, $\mathcal{C} = \{C_j\}$, of subsets of K, where each C_j is a set of pairwise connected kernels. A subset $L \subseteq K$ is connected if, for any pair of kernels $k, k' \in L$, there is a sequence $k = k_1, k_2, \cdots, k_n = k'$, with each $k_i \in L$ and each pair of consecutive kernels, k_i and k_{i+1}, connected by being members of some C_j. The whole set of kernels, K, should be connected.

The output of the algorithm is two map functions. Firstly, T maps kernels onto tasks, and secondly, P maps tasks onto processors. The partition implied by T must be convex, so the graph of dependencies between tasks is acyclic.

Let $T_p = P^{-1}(p)$ be the tasks on processor p, and $K_t = T^{-1}(t)$, $K_p = \bigcup_{t \in T_p} K_t$ be the kernels on task t or processor p. The graph of t is the induced graph, $G_t = G(K_t)$, containing the kernels in t and internal streams. The task dependence graph G^T is the result of contracting each task in G into a single vertex.

The cost on processor p or interconnect u is:

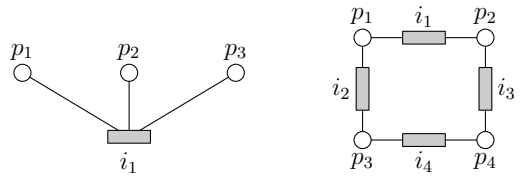
$$C^p = \sum_{i \in K_p} \frac{c^{ip}}{w^p}$$

$$C^u = \sum_{p,q \in P} r^u_{pq} \sum_{i \in K_p, j \in K_q} \frac{c^{ij}}{w^u}$$

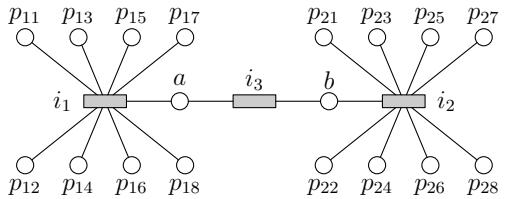
The goal is to find the allocation (T, P), which minimises the maximum values of all the C^p and C^u, subject to the convexity and connectedness constraints.

4.1 Predicting memory use of tasks

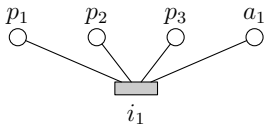
When multiple kernels are fused into one task, we need to predict the memory use of the task, given the memory use and composition of each kernel. Finding the minimum memory use is an $\mathcal{NP}$-complete problem [3], even ignoring the possibility to overlap the buffers for two or more streams. Not only that, but the partitioning algorithm needs to pre-



(a) SMP with three processors *(b) 2×2 mesh*



(c) IBM QS20 with two Cell processors (SPEs only)



(d) SMP 3 with accelerator (a_1)

Figure 6: Topology of the targets used in this paper (interconnects are shown as shaded rectangles)

dict, or at least bound, the memory use from the actual compiler, which is unlikely to be the theoretical minimum.

In this paper, we assume that the combined code size is the sum of the code sizes for the kernels, and that the total memory size is one block, plus the length of the history, for each internal stream, and the original memory size for each external stream. This calculation is orthogonal to the rest of the algorithm.

5. DESCRIPTION OF THE ALGORITHM

The partitioning algorithm is split into two phases. The first phase produces an initial partition that is both convex and connected, with at most one task per processor. The second phase, refinement, improves the initial partition, and has some ability to escape from local minima; it can also create multiple tasks per processor.

The first phase could produce a trivial initial partition, which has all kernels in a single task, assuming enough memory. Our results show that the refinement phase still finds a good partition. A good initial partition, however, decreases the total time of the mapping algorithm, since it requires fewer passes of the refinement phase.

The refinement phase uses several algorithms based on Kernighan and Lin's graph partitioning algorithm [19], and is repeated until there is no further improvement. The main step offloads kernels from bottleneck processors, while maintaining connectedness and convexity. If this produces no benefit, then one additional task is created, if enabled, and the new partition is kept if the improvement is larger than some threshold (currently 5%).

5.1 Initial partition

The initial partition is generated by recursively subdividing the target and program graphs into halves, mapping each half separately. This continues until there is either a single kernel, which is mapped to some processor, or a single processor, which executes all kernels.

Partitioning the target

The algorithm first divides the target into two subgraphs, P_1 and P_2, and an aggregate interconnect, I, balancing two objectives: the subgraphs should have roughly equal total CPU performance, and the aggregate interconnect bandwidth between them should be low. Figure 7(a) shows the result of dividing the mesh target from Figure 6(b).

The optimal target partition is found as follows. First, the communications bottleneck for uniformly random traffic between P_1 and P_2 is given by:

$$C = \max_{u \in I} \sum_{p \in P_1, q \in P_2} \frac{r^u_{pq} + r^u_{qp}}{w^u} \tag{1}$$

The target is divided into halves to maximise α, the product of C with the total performance of the less powerful of P_1 or P_2:

$$\alpha = C \min \left(\sum_{p \in P_1} w^p, \sum_{q \in P_2} w^q \right) \tag{2}$$

We find an approximate solution using a variant of the Kernighan and Lin partitioning algorithm.

Partitioning the program

The program (sub)graph is given edge and vertex weights. The edge weight for stream ij, denoted c_{ij} is the cost in cycles in time τ, if assigned to the aggregate interconnect, rather than internal to P_1 or P_2. The vertex weight for kernel i is a pair (c^{iP_1}, c^{iP_2}), the cost of assigning it to P_1 or P_2, respectively. The goal is to find a two-way partition $\{K_1, K_2\}$ to minimize the bottleneck given by:

$$c = \max \left(\sum_{i \in K_1} c^{iP_1}, \sum_{j \in K_2} c^{jP_2}, \sum_{i \in K_1, j \in K_2} c_{ij} \right) \tag{3}$$

The partitioning algorithm is a branch and bound search. Each node in the search tree inherits a partial partition (K_1, K_2), and unassigned vertices X; at the root $K_1 = K_2 = \phi$ and $X = K$. It chooses some kernel $v \in X$, adjacent to K_1 with $K_1 \cup \{v\}$ convex and connected (or any v if K_1 is empty) then switches on either adding v and its ancestors to K_1, or v and its descendants to K_2. If adding vertices to K_1 would cause $K_2 \cup X$ to become disconnected, then the subtree contains no connected partitions, so is pruned.

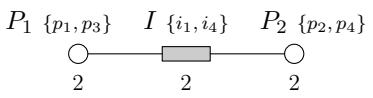
Figure 7(b) and (c) show a program and its branch and bound search, with each node labelled by its sets K_1 and K_2. The minimal cost, $c^{K_1K_2}$, for all partitions in the subtree rooted at node (K_1, K_2) is at least as large as the partial sum on the vertices already assigned:

$$c^{K_1K_2} \geq lb = \max \left(\sum_{i \in K_1} c^{iA}, \sum_{i \in K_2} c^{iB}, \sum_{i \in K_1, j \in K_2} c_{ij} \right) \tag{4}$$

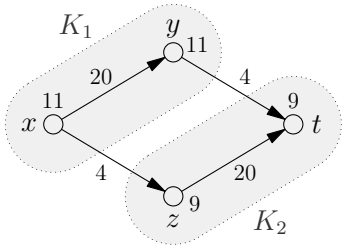
Any valid partition in the subtree gives an upper bound on the optimal cost in that subtree. Since $(K_1, X \cup K_2)$ is always valid:

$$c^{K_1K_2} \leq ub = \max \left(\sum_{i \in K_1} c^{iA}, \sum_{i \notin K_1} c^{iB}, \sum_{i \in K_1, j \notin K_1} c_{ij} \right) \tag{5}$$

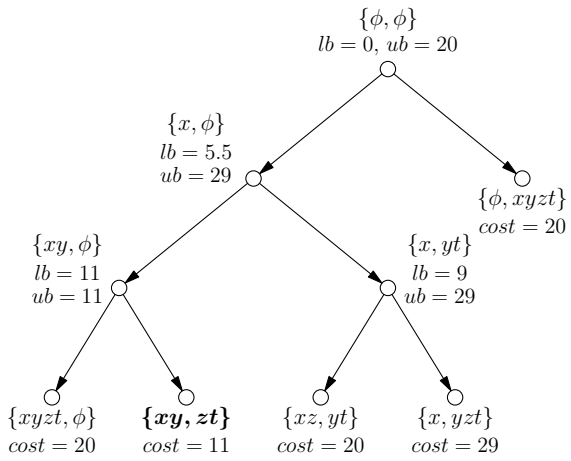
In Figure 7(c), the node marked $\{x, \phi\}$ has $K_1 = \{x\}$ and $K_2 = \phi$. The known cost on P_1 is 5.5, being the cost of kernel x divided by the performance of P_1. The known costs on P_2



(a) First level partition of the target in Figure 6(b)



(b) Stream program and first level partition



(c) Branch and bound search for first level partition

Figure 7: First level partition in the initial partition algorithm

and the interconnect are both zero. Hence by Equation (4), $lb = \max(5.5, 0, 0) = 5.5$. Similarly, by Equation (5), the upper bound on the optimum is the cost of partition $\{x, yzt\}$, so $ub = 29$.

The search algorithm tries to quickly find a good partition, so that more of the search tree is pruned by having its lower bound greater than some upper bound. It uses a depth-first search, and chooses vertex v adjacent to K_1 (as it must) with the highest cost on whichever processor currently has the greatest load, then first considers adding it to the other processor.

5.2 Refinement of the Partition

The refinement stage starts with a valid initial partition, and improves it by applying the optimization passes described below. As shown in Figure 1, these steps are applied in sequence, and iterated until no further improvement is seen. The optimization passes are:

Merge tasks A greedy algorithm merges low cost tasks and has the effect of freeing processors and reducing communications

Move bottlenecks The main optimization pass moves kernels from bottleneck processors

Create tasks Create a new task to relax the connectedness and convexity constraints, and keep the new partition if the benefit is larger than some threshold

Reallocate tasks A greedy algorithm improves the allocation of tasks to processors

The passes are described in detail below.

Merge tasks

This step uses a greedy algorithm to merge tasks whose union is convex and connected, as long as it does not cause a new bottleneck. This pass often reduces bus traffic, and frees up processors so they can accept kernels without restriction. Since there are usually far fewer tasks than kernels, we define the basic connected sets of tasks: $D_j = \{T(k) : k \in C_j\}$, where $T(k)$ was defined earlier as the task containing kernel k, and set $\mathcal{D} = \{D_j : |D_j| \geq 2\}$. In this case, the union of T_1 and T_2 is connected if $\{T_1, T_2\} \subseteq D_j$, some $D_j \in \mathcal{D}$.

In Section 4, we defined the task dependence graph, G^T, as the directed acyclic graph on the tasks. Define $d(T_1, T_2) = 1$ if there is a path from T_1 to T_2 of length two or more, and 0 otherwise. This can be calculated in time $O(|T|^2)$, using a topological sort. The greedy algorithm finds, using a branch and bound search, the connected pair of tasks T_1 and T_2 with minimum total cost on either of their current processors, such that $d(T_1, T_2) = 0$. If the bottleneck cost after merging is no greater than the current bottleneck cost, then the tasks are merged and allocated to the processor on which they have the minimum total cost. The algorithm continues until no more tasks can be merged.

Move bottlenecks

This pass identifies a bottleneck processor, p_1, then considers moving a set M_1 of kernels from some task on p_1 to a task on another processor, q_1, without violating convexity or connectedness. The cost metric to minimise is the maximum of the costs on p_1, q_1 and all interconnects, after the move:

$$C = \max(C^{p_1}, C^{q_1}, \max_{u \in I} C^u) \tag{6}$$

This metric excludes the other processors, otherwise if some other processor had the same cost as p_1, its contribution would hide the benefit of any move.

Some kernels must be moved, even if doing so has a negative benefit—hence the algorithm has some ability to escape from local minima. After tentatively moving set M_1, record the bottleneck cost and identify the new bottleneck processor, p_2, which may still be p_1, and tentatively move a set M_2 to another processor. We continue moving kernels, with the constraint that no kernel can be moved back to a processor that it has previously been allocated to. For instance, none of the kernels in M_1 may be tentatively moved back to p_1, but they may be moved a second time to another processor. This process continues until either there are no remaining valid moves, or a fixed limit, currently 50 moves, is reached. The final partition is that of the intermediate point in the algorithm with the maximum overall performance.

Any kernel, k, can potentially be moved to any task on a different processor if there is a kernel k' on the new task that shares a basic connected set with k. There are three additional requirements. Firstly, if k is neither a source nor a sink in its current task, T, then $T - \{k\}$ cannot be convex. It is always necessary to move either k and all its ancestors in T, or k and all its descendants in T. Secondly, it is necessary to check, using a breadth-first or depth-first search on the basic connectivity sets, whether the remainder of the old task is still connected. Thirdly, there are several ways that the move can create a cycle in the task dependency graph, and this can be checked using a topological sort.

Create tasks

This pass moves a kernel k on a bottleneck processor onto another processor, creating new tasks as necessary to become convex and connected. It then runs the Move Bottlenecks pass, with the restriction that the kernel cannot be moved from its new processor. The new partition is kept if the performance is improved by more than some threshold, currently 5%.

The most expensive kernel, on its current bottleneck processor, is considered first. This kernel may be moved to any processor in use for which the cost of the kernel is less than the current bottleneck cost. There is no advantage in moving a kernel to an unused processor, since that is the first thing that the Move Bottlenecks pass would do. If there are no valid choices, then the second most expensive kernel on the same processor is considered, and so on.

Kernel k is placed on some other processor in order to minimize the sum of the weights of the *large kernels*, including k, on the new processor; the large kernels are those of weight at least half that of the kernel being moved. The reason for ignoring lightweight kernels is that these are most likely to be able to be easily offloaded onto other processors.

Reallocate tasks

The reallocation pass decreases communications traffic by permuting the loads on the processors. This pass is executed even if the bottleneck is on one of the processors. It only moves tasks between similar processors attached to different buses. Similar processors are those for which all kernel computation times are identical. For instance, it can permute the four processors on the 2×2 mesh target, or SPEs on different processors on the two-Cell QS20 target.

The algorithm is similar to Kernighan & Lin, in that it swaps similar processors in a greedy manner to minimize the maximum load on the buses, even if doing so makes the bottleneck worse. After swapping the loads on two processors, they are fixed for the rest of the pass. The algorithm continues until there are no processors left, and outputs the best partition seen.

6. EVALUATION

We used the StreamIt 2.1.1 benchmarks [11] to evaluate our heuristic algorithm and convex connected partitions in general. The StreamIt benchmarks have the two-terminal series-parallel structure of StreamIt, but are the most widely used streaming benchmarks. We used the program graph, work estimates and data rates from the StreamIt 2.1.1 compiler. The StreamIt compiler modifies the stream program graph before calculating the work estimates, so our kernel counts differ from those of the source program. The number of kernels ranges from 8 to 120, and has average 54.

Figure 8 shows performance vs. iteration in the refinement phase. At each point is plotted the minimum of all partitions seen so far. This graph shows that the refinement algorithm quickly converges to a good solution, even from a trivial initial partition. It also shows that the initial partition is on average within 4% of the performance of the final partition, although the worst case is 33% slower. The sub-figures have very different scales on the vertical axes. We show the number of iterations in the graphs, since our implementation is in unoptimised Python. Nevertheless, the per benchmark partitioning time on a 2GHz Intel MacBook is average 10.0 seconds and maximum of 58.4 seconds.

Figure 9 shows the normalized execution time for the partitions found by the heuristic, using strict and loose connectivity, against the optimal unrestricted partition, which has time 1.0. The strictly connected partitions for *channel*, *filterbank*, *fm* and *radar* have bad performance because of the wide split joins. The third column of Table 1 gives the width of each benchmark, which is the maximum size of a subset of kernels with no paths between any pair of them (an anti-chain). For example, the *filterbank* benchmark, illustrated in Figure 2, has a width of 16. The benchmarks with poor performance using strict connectivity tend to be the benchmarks with the largest width, although this is a great simplification.

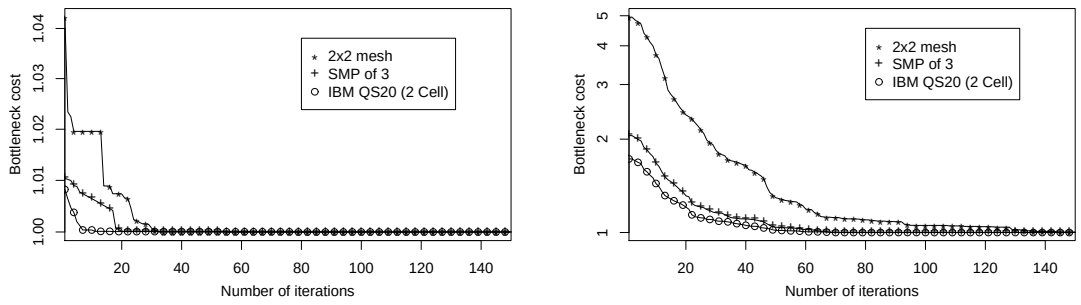
Figure 9 also shows the bottleneck cost for the loosely connected partition generated by our heuristic when software pipelining is enabled. Software pipelining is most beneficial for the *vocoder* benchmark on SMP3 with accelerator, and *radar* on IBM QS20. Figure 10 shows the partition of *vocoder* on SMP with accelerator found using the heuristic, which uses eleven pipeline stages if scheduled using the stage assignment phase of the SGMS algorithm [20]: six for computation and five for DMA. The main improvement in throughput comes from splitting the workload on a_1 into two tasks. This benchmark has two heavyweight kernels that should run on the accelerator, but the cost of the smallest convex task containing both of them is very large. The optimal unrestricted partition is 9% faster, but it requires 45 pipeline stages.

The only bad results are *des*, *serpent*, and *tde* on IBM QS20. These three suffer since our fast greedy algorithm gets stuck in a local optimum, with several large tasks of similar total cost.

Table 1 shows the pipeline lengths for the unrestricted and heuristic partitions. The pipelines were generated using SGMS [20], so half of the pipeline stages perform computation. The partitions that do not require software pipelines are given in parentheses. Some of the cells for the unrestricted partitions are empty, since finding the true optimum is slow (Figure 9 uses a lower bound). The unrestricted partitions for *serpent* and *tde* on Cell have long software pipelines because the programs themselves are long pipelines, and unless it is bus bound, there is no incentive for short pipelines. There are four data points in Figure 9 where our heuristic algorithm used software pipelining.

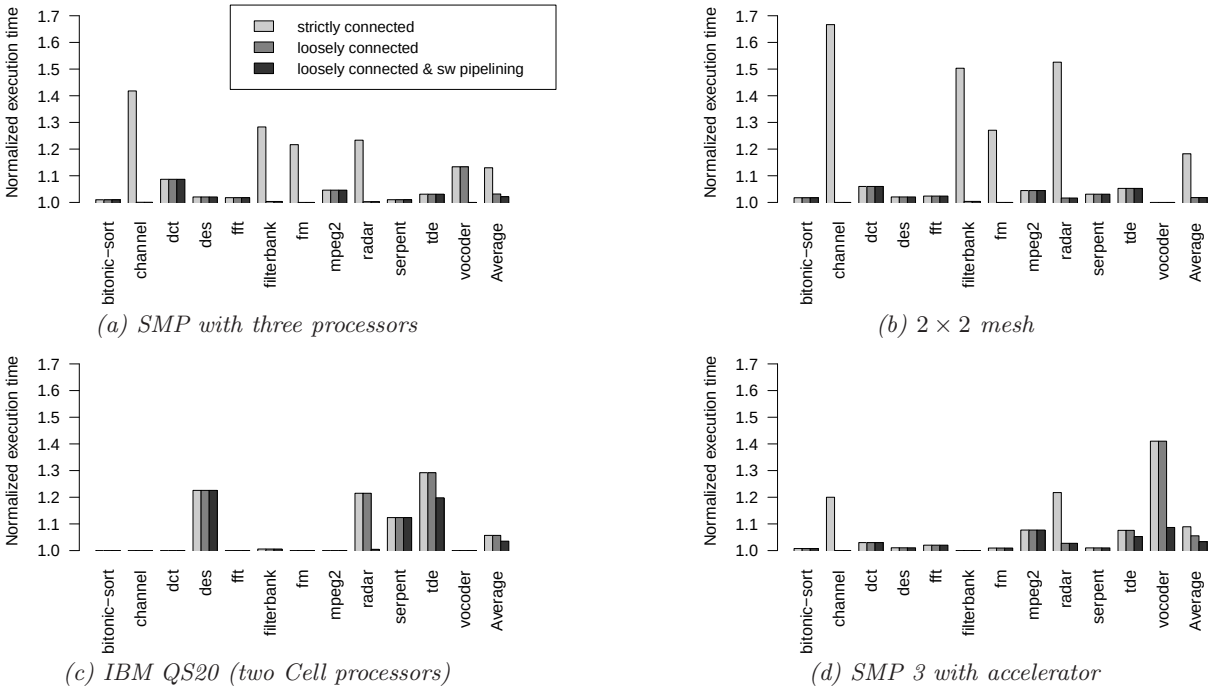
7. RELATED WORK

There has been a great deal of work in automatically mapping stream programs onto multiprocessor systems. The Ptolemy II software environment [10] is an actor-based model



(a) Recursive initial partition *(b) Trivial initial partition*

Figure 8: Convergence of the refinement phase as a function of the number of iterations



(a) SMP with three processors *(b) 2×2 mesh*

(c) IBM QS20 (two Cell processors) *(d) SMP 3 with accelerator*

Figure 9: Normalized execution time for the StreamIt 2.1.1 benchmarks for the three variants of the heuristic algorithm. The unrestricted partition has execution time 1.0, and larger bars are slower.

Benchmark	Num. kernels	Width	Number of pipeline stages for unrestricted & heuristic							
			SMP 3		2×2 Mesh		QS20 (2 Cell)		SMP 3 + Acc.	
			u	h	u	h	u	h	u	h
bitonic-sort	40	4	27	(5)	35	(7)	35	(19)	9	(7)
channel	55	17	11	(5)	9	(7)	9	(7)	9	(5)
dct	8	1	9	(5)	9	(7)	9	(9)	13	(5)
des	53	3					77	(31)		
fft	17	1	13	(5)	27	(7)	33	(23)	17	(7)
filterbank	85	16	19	(5)	19	(7)	23	(9)	19	(5)
fm	43	12	9	(5)	17	(5)	21	(9)	17	(5)
mpeg2	23	5	23	(5)	19	(7)	29	(19)	15	(7)
radar	57	12	13	(5)	9	(5)	19	13	19	(5)
serpent	120	2	143	(5)	163	(7)	209	(31)		
tde	29	1	39	(5)	41	(7)	57	33	23	(9)
vocoder	114	17	29	7	29	(7)	29	(7)	45	11
Average ratio			5.9		5.0		2.4		2.8	

Table 1: The number of pipeline stages for the optimal unrestricted partitions and the partitions generated by our heuristic (loosely connected with pipelining). Partitions that do not need pipelining are in parentheses.

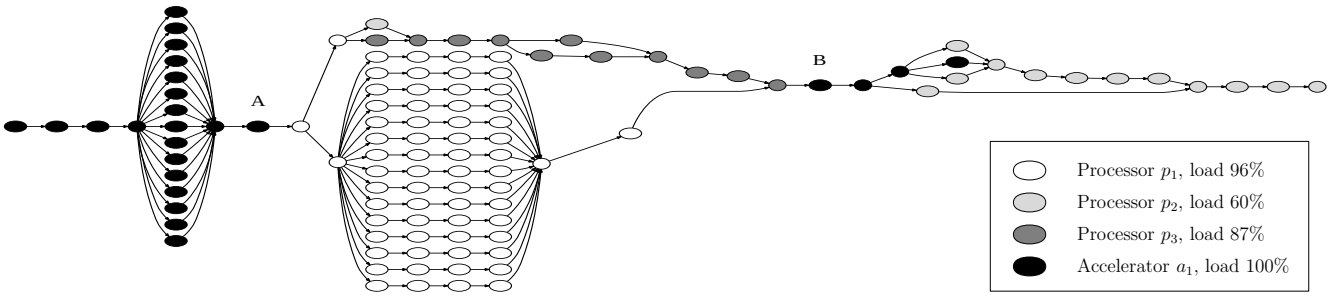


Figure 10: Vocoder benchmark on SMP 3 with accelerator using software pipelining.

for real-time embedded systems that supports several models of computation, including Synchronous Dataflow (SDF) and Kahn Process Networks (KPN). Related work from the Ptolemy project explores the more theoretical aspects of partitioning and scheduling data flow graphs for multiprocessors [13].

The Stream Graph Modulo Scheduling (SGMS) algorithm is part of StreamRoller [20], a StreamIt compiler for the Cell Architecture. This algorithm splits stateless kernels, partitions the graph, and statically schedules. The splitting and partitioning problem is translated into an Integer Linear Programming (ILP) problem, which is solved using CPLEX [15]. This approach uses mature technology to solve the ILP problem; it also applies kernel splitting in the same step, rather than using the iterative approach we follow.

Their partitioning algorithm considers only CPU loads, and ignores communications bandwidth. This may be sufficient for a single Cell processor, which has a high-bandwidth on-chip bus, but it is inappropriate when communication is off-chip, as in the Cell QS20 target, or when a bottleneck may appear in part of an on-chip network, such as a large mesh.

The StreamRoller ILP formulation does not attempt to find a partition that minimises the memory, latency and startup overheads introduced by software pipelining. Since it uses an ILP solver to find a (close to) optimal solution to a problem with similar objective and constraints to our unrestricted partition, the resulting pipeline length should be similar. StreamRoller does not have any concept similar to our connectivity constraint. We believe that when the program is written using an unrestricted programming language, the partitioning algorithm requires some mechanism to model which kernels can be statically scheduled by the compiler. They do not restrict the memory footprint on each processor, although it appears that their ILP formulation could be extended to do so.

The StreamIt compiler [12, 11] targets the Raw Microprocessor [29], symmetric multicore architectures, and clusters of workstations. This is a long running project with a publicly available compiler and benchmark suite. The StreamIt source language imposes a structure on the stream program graph, where each kernel has a single input and a single output, and kernels are composed in pipelines, split-joins, and feedback loops. Since the kernels have static data rates, the compiler can fuse any set of kernels. The default partitioner uses dynamic programming. Our model of the source program is more general, since we target unstructured program graphs with variable data rates, and we use the connectedness constraint to reason about the capabilities of the compiler. Our model of the target system is also more general, since we can target a heterogeneous multiprocessor system with any communications topology.

Liao et al. [23] use affine partitioning to map regular multidimensional programs written using the Brook language [5] onto a four-processor SMP. The R-Stream compiler (www.reservoir.com/r-stream.php) is a proprietary high level compiler for stream programs, which uses a polyhedral model to partition code and data to a parametric parallel machine. GEDAE [24] is a proprietary GUI tool for mapping data flow graphs to a heterogeneous multiprocessor system. The transformations are under user control, and the partition is not automatically found by the compiler.

8. CONCLUSIONS

In this paper, we presented a fast and robust partitioning algorithm for an iterative stream compiler. The algorithm maps an unstructured variable data rate stream program onto a heterogeneous multiprocessor system with any communications topology. The algorithm favours convex connected partitions, which do not require software pipelining and are easy to compile, but it can break these constraints when necessary for performance. The partitions generated by the heuristic seldom require software pipelines, and the performance is, on average, within 5% of the optimum performance.

9. ACKNOWLEDGEMENTS

The researchers at BSC-UPC were supported by the Spanish Ministry of Science and Innovation (contract no. TIN-2007-60625), the European Commission in the context of the ACOTES project (contract no. IST-34869) and the HiPEAC Network of Excellence (contract no. FP7/ICT 217068). We would also like to acknowledge our partners in the ACOTES project for the insightful discussions on the topics presented in this paper.

10. REFERENCES

[1] ACOTES. IST ACOTES Project Deliverable D2.2 Report on Streaming Programming Model and Abstract Streaming Machine Description Final Version. 2008.

[2] A. Artieri, VD Alto, R. Chesson, M. Hopkins, and M.C. Rossi. Nomadik Open Multimedia Platform for Next-generation Mobile Devices. *STMicroelectronics Technical Article TA305*, 2003.

[3] S.S. Battacharyya, P.K. Murthy, and E.A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

[4] M. Becchi and P. Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the 3rd conference on Computing frontiers*, pages 29–40. ACM New York, NY, USA, 2006.

[5] I. Buck. Brook Spec v0.2, 2003.

[6] Paul M. Carpenter, Alex Ramirez, and Eduard Ayguade. The Abstract Streaming Machine: Compile-Time Performance Modelling of Stream Programs on Heterogeneous Multiprocessors. In *SAMOS Workshop 2009*, pages 12–23. Springer.

[7] J. Chaoui, K. Cyr, J.P. Giacalone, S. Gregorio, Y. Masse, Y. Muthusamy, T. Spits, M. Budagavi, and J. Webb. OMAP: Enabling Multimedia Applications in Third Generation (3G) Wireless Terminals. *SWPA001, December*, 2000.

[8] A. Cohen, S. Girbal, and O. Temam. A polyhedral approach to ease the composition of program transformations. *Lecture Notes in Computer Science*, pages 292–303, 2004.

[9] S. Dutta, R. Jensen, and A. Rieckmann. Viper: A Multiprocessor SOC for Advanced Set-Top Box and Digital TV Systems. *IEEE Design & Test of Computers*, pages 21–31, 2001.

[10] J. Eker, J.W. Janneck, E.A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity–the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[11] M.I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 151–162, 2006.

[12] M.I. Gordon, W. Thies, M. Karczmarek, J. Lin, A.S. Meli, A.A. Lamb, C. Leger, J. Wong, and H. Hoffmann. A Stream Compiler for Communication-Exposed Architectures. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–303, 2002.

[13] S. Ha and E.A. Lee. Compile-time scheduling and assignment of data-flow program graphs with data-dependent iteration. *Computers, IEEE Transactions on*, 40(11):1225–1238, Nov 1991.

[14] H.P. Hofstee. Power Efficient Processor Architecture and The Cell Processor. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 258–262. IEEE Computer Society Washington, DC, USA, 2005.

[15] ILOG. http://www.ilog.com/products/cplex/. CPLEX Math Programming Engine.

[16] Intel. *IXP2850 Network Processor: Hardware Reference Manual*, 2004.

[17] ACOTES IST-034869. http://www.hitech-projects.com/euprojects/ACOTES/. Advanced Compiler Technologies for Embedded Streaming.

[18] G. Kahn. The semantics of a simple language for parallel processing. *Information Processing*, 74:471–475, 1974.

[19] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, 1970.

[20] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *Proc. 2008 ACM SIGPLAN Conference on Programming Language Design & Implementation*, pages 114–124. ACM New York, NY, USA, 2008.

[21] R. Kumar, DM Tullsen, NP Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32–38, 2005.

[22] EA Lee and DG Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[23] S. Liao, Z. Du, G. Wu, and G.Y. Lueh. Data and Computation Transformations for Brook Streaming Applications on Multiprocessors. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 196–207, 2006.

[24] W.I. Lundgren, K.B. Barnes, and J.W. Steed. Gedae: Auto Coding to a Virtual Machine. In *8th High Performance Embedded Computing Workshop*, 2004.

[25] Joseph Muscat and David Buhagiar. Connective Spaces. *Mem. Fac. Sci. Eng. Shimane Univ. Series B: Mathematical Science*, 39:1–13, 2006.

[26] Kunle Olukotun and Lance Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005.

[27] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[28] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *11th International Conference on Compiler Construction*, pages 179–196. Springer, 2002.

[29] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, et al. Baring It All to Software: Raw Machines. *Computer*, pages 86–93, 1997.