# Buffer sizing for self-timed stream programs on heterogeneous distributed memory multiprocessors

Paul M. Carpenter, Alex Ramirez, and Eduard Ayguade

Barcelona Supercomputing Center, C/Jordi Girona, 31, 08034 Barcelona, Spain
{paul.carpenter,alex.ramirez,eduard.ayguade}@bsc.es

**Abstract.** Stream programming is a promising way to expose concurrency to the compiler. A stream program is built from kernels that communicate only via point-to-point streams. The stream compiler statically allocates these kernels to processors, applying blocking, fission and fusion transformations. The compiler determines the sizes of the communication buffers, which affects performance since local memories can be small.
In this paper, we propose a feedback-directed algorithm that determines the size of each communication buffer, based on i) the stream program that has been mapped onto processors, ii) feedback from an earlier execution, and iii) the memory constraints. The algorithm exposes a trade-off between throughput and latency. It is general, in that it applies to stream programs with unstructured stream graphs, and it supports variable execution times and communication rates.
We show results for the StreamIt benchmarks and random graphs. For the StreamIt benchmarks, throughput is optimal after the first iteration. For random graphs with stochastic computation times, throughput is within 3% of optimal after four iterations. Compared with the previous general algorithm, by Basten and Hoogerbrugge, our algorithm has significantly better performance and latency.

## 1 Introduction

Many applications, including video, audio, 3D graphics, and radio, contain abundant task and data parallelism, but it is hard to extract from C source code. Stream programming represents the application as concurrent kernels, interacting only via point-to-point streams of data. This representation exposes concurrency to the compiler, is natural for signal processing, and easier to debug since it is deterministic. As the industry moves towards multiprocessors [1], there is increasing interest in portable, efficient, correct use of parallelism.

Much work on stream compilation has focused on blocking and allocation. Blocking unrolls kernels to amortise fixed costs. Allocation fuses one or more kernels, from the source program, into each task, in the executable, and maps these tasks onto processors, balancing loads on processors and buses.

This paper considers a problem that has received less attention: allocating memory for stream buffers, subject to memory constraints, when computation

*To appear in* HiPEAC 2010, LNCS, 2010

times and communication rates are variable. This is an important problem, because it affects performance, as we explain in Section 2. The buffer sizes are constrained by the available memory, which may be small. On the Cell Broadband Engine [2], for example, code and data must fit in the 256KB local store.

The inputs to the algorithm are the *mapped stream program*, a *program trace* and the *machine description*, giving the target topology and memory budgets. A simple model of computation times and communication rates, such as independent normal distributions and Poisson arrivals, may be misleading, so the only options are simulation and real execution. We use coarse-grain simulation, but real execution could be used instead. The output is the buffer size for the producer and consumer on each stream, which may be different.

The main contributions of this paper are:

– In Section 3, we describe a feedback-driven method to allocate stream buffers in a distributed memory machine, when computation times and communication rates are variable.
– In Section 5.1, we describe two algorithms that analyse profiling information to find bottleneck cycles caused by undersized communication buffers. The first uses waiting times only; the second is more complex but more accurate.
– In Section 5.2, we describe an algorithm to allocate stream buffers using the above algorithms, which converges quickly to a close-to-optimal allocation.

## 2 Motivation

Double buffering is a well-known technique to overlap communication and computation. There are two situations, however, when a stream ought to be allocated more than two buffers. The first is when a stream covers a long latency or, equivalently, crosses more than one pipeline stage boundary. The second is when there are short-duration load imbalances due to variable computation times or communication rates.

The *chain8* benchmark illustrates the first situation, and is shown on the left of Figure 1. It has eight tasks in a pipeline, with streams between consecutive tasks, and another stream between the first and last tasks. Figure 1(a) shows the progress of the first and last tasks relative to the stream between them. The vertical axis is time, and the horizontal axis is the position in the stream. At any given time the producer is working on some interval of the stream, which it owns. It starts at the top left of the plot, at the beginning of both the stream and time, moving to the right when it sends data to the consumer, and continually downward through time. The figure also shows the progress of the consumer.

The periodic pattern of waiting is caused by the interaction between two dependencies. First, the consumer must wait for its data to arrive, which means that it waits for the producer, plus the latency of the pipeline. This gives a vertical dependency from producer to consumer. Second, the producer must wait for an empty consumer-side buffer in which to send its data, and this gives a horizontal dependency from consumer to producer.

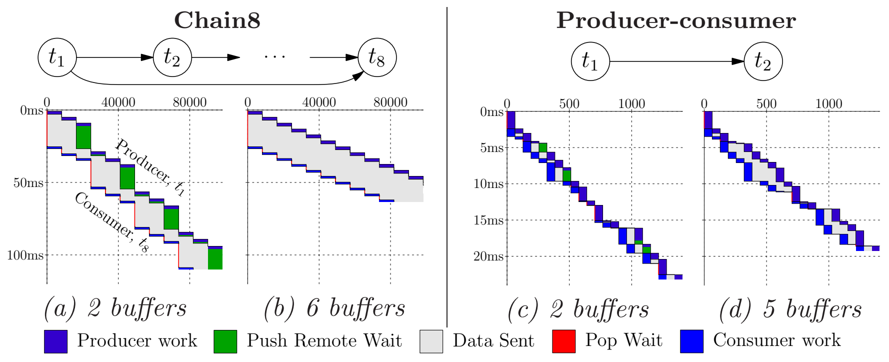


**Fig. 1.** Effect of consumer queue length on chain8 and producer-consumer

Figure 1(b) is for six consumer-side buffers, which increases throughput by 73%, and is sufficient for the producer to be always busy. This shows that double buffering was not sufficient, but also that the number of buffers can be less than one plus the difference in pipeline stage, which is the number of buffers allocated by StreamRoller [3] and SPIR [4]; in this case eight.

The second situation is illustrated using the *producer-consumer* example on the right of Figure 1. If the producer and consumer both have fixed computation times and communication rates, then double buffering is sufficient. Sometimes, single buffering at one or other end will be enough, even with good load balancing. Subfigure (c) shows the progress of this example, using double buffering, when computation times are normally distributed. Increasing the number of consumer buffers to five, as shown in subfigure (d), increases throughput by 20%.

The performance of the queue length assignment algorithm is quantified using the *utilisation*, which is the percentage of time that the most heavily loaded processor or bus is busy. Utilisation is proportional to throughput. If the stream graph is acyclic, at least one resource ought to be 100% busy. If any resource has utilisation less than 100%, it must be due to insufficient buffering.

The tradeoff between utilisation and the number of consumer buffers is illustrated in Figure 2. Chain has linearly increasing utilisation until it reaches 100%. Producer-consumer achieves 99% utilisation with 3 producer and 4 consumer buffers, and additional buffering yields diminishing returns.

## 3 The ACOTES Stream Compiler

This work is part of the ACOTES European project [5], which is developing an open source stream compiler for embedded systems. The compiler will map a portable stream program, written in the SPM [6], an annotated version of C, onto a heterogeneous multicore system, applying blocking and task fusion.

The compiler statically allocates tasks onto processors. Although a dynamic policy can achieve better load balance [7], it has greater overhead. On a distributed memory processor, instructions and state cannot be transferred on demand through caches, so a context switch requires all data to be transferred at once. A context switch on the Cell SPE requires about $30\mu$s [8]. The techniques in this paper can be used to absorb small scale variation in complexity.

Figure 3 shows how the queue length assignment algorithm fits into this stream compiler. The blocking and partitioning stages transform the program as described in the introduction. The queue length assignment stage, which is the focus of this paper, then determines the optimal buffer allocation.

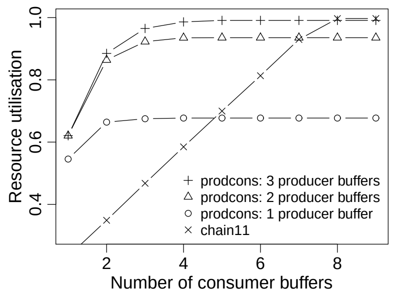


**Fig. 2.** Memory-performance tradeoff

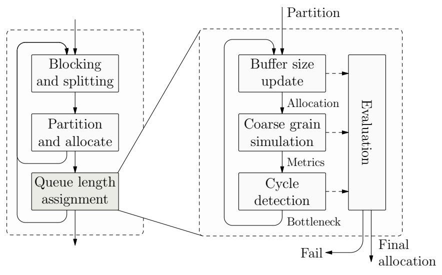


**Fig. 3.** Mapping phase of the compiler

Our SPM language eliminates deadlock, so the objective function depends only on performance and latency. The interaction between bounded memory in process networks and deadlock, but not performance, has been explored in depth [9–11], and these techniques can determine the *minimum* buffer sizes.

The queue length assignment algorithm is iterative, and consists of a coarse-grain simulator, a *cycle detection algorithm*, a *buffer size update algorithm*, and an *evaluation algorithm*. The cycle detection algorithm analyses metrics from the simulator, and finds a bottleneck cycle. The buffer update algorithm chooses the initial buffer allocation, and adjusts buffer sizes to resolve the bottleneck. The evaluation algorithm monitors progress and decides when to stop, choosing the buffer allocation that achieved the best performance-latency tradeoff.

## 4 Formalisation of the Problem

Queue length assignment seeks to find an optimal tradeoff, subject to memory constraints, between throughput and latency We wish to find a close to Pareto optimal solution: that is, neither latency nor throughput can be improved without making the other one worse. We keep memory use within the constraints, but do not try to minimise it.

The stream program is represented as a connected, not necessarily acyclic, digraph, $\mathcal{P} = (T, S)$, where $T$ is the set of vertices (tasks), and $S$ is the set of edges (streams). Each stream $s$ has a producer and consumer buffer size in bytes, $b_p(s)$ and $b_c(s)$, and a minimum number of buffers, sufficient to hold the working set and avoid deadlocks. If $P$ is acyclic, as for ACOTES, deadlock is impossible; otherwise minimum sizes can be found using the references in Section 3. The algorithm determines the actual number of buffers, $n_p(s)$ and $n_c(s)$.

Each task has a trace, which is an alternating sequence of computation times and primitives. There are four communications primitives and a fire primitive, which marks the firing of a task; i.e. the calling of its work function inside an implicit loop. The communications primitives use a push model similar to the DBI variant of TTL [12]. They are described below, assuming, for simplicity, that the producer and consumer have the same buffer size, which is not required. A *block* is the contents of one buffer, and $i$ and $j$ count blocks, starting at zero. The first argument is the stream.

**ProducerAcquire(s, k)** Wait for the producer buffer for block $i + k$ to be available, meaning that the DMA transfer of block $i + k - n_p(s)$ has completed

**ProducerSend(s)** Wait for the consumer buffer for block $i$ to be available, meaning that the producer has received acknowledgement that block $i - n_c(s)$ has been discarded. Then send the block and increment $i$

**ConsumerAcquire(s, k)** Wait for block $j + k$ to arrive in the consumer buffer

**ConsumerDiscard(s)** Discard block $j$, send acknowledgement, and increment $j$

The traces are interpreted using the ASM coarse-grain simulator, which takes a machine description that defines the target [13]. Queue length assignment needs only the memory constraints, which are represented using a bipartite graph, $\mathcal{H} = (R, E)$. The set of vertices, $R = P \cup M$, is a disjoint union of processors $P$ and memories $M$, and the edges, $E$, connect processors to their local memories. Each memory has weight equal to the amount of memory available, in bytes, for stream buffers. Figure 4 shows the memory constraint graph for the Cell Broadband Engine; the memory weights depend on how much memory is already being used. We will later assume that each processor is connected to a single memory, but it may be shared with other processors.

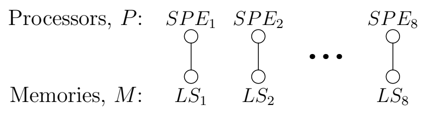


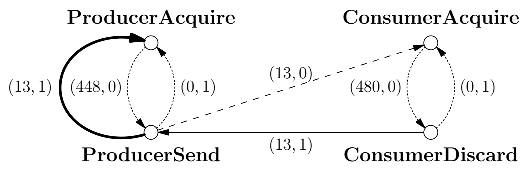
**Fig. 4.** Memory constraint graph for the Cell Broadband Engine

The evaluation algorithm and Section 6 of the paper require an estimate of latency. Since it is orthogonal to the rest of the paper, and only differences in latency matter, we use a scheme which ignores delays inside tasks.

Define $f_t(n)$ to be the time of firing, $n = 0, 1, \cdots, M_t - 1$ of task $t$, taken from the fire primitive. Since each task contributes to a common amount of real-world progress, normalise $n$ to the interval $0 \leq x < 1$ by dividing it by $M_t$. Then $g_t(x) = f_t(\lfloor M_t x \rfloor)$ gives the time that task $t$ was proportion $x \in [0, 1)$ through the calculation. The latency, $L(x)$, is the difference between the largest $g_t(x)$ for a sink and the smallest $g_t(x)$ for a source, which can, unfortunately, be negative when multiplicities are variable. We report the average value of $L(x)$.

## 5 Description of the Algorithms

In this section, we describe several algorithms for cycle detection and buffer size update. First we review the standard critical cycle detection algorithm, and explain when it is applicable. We introduce our *baseline* algorithm, which finds the bottleneck cycle by analysing the time each task is blocked on each stream. This data is easy to obtain, and the algorithm is quite effective. We then give an example that the baseline algorithm gets wrong, and propose the *token* algorithm, which requires extra bookkeeping but achieves better results. Finally, we describe several variants on the buffer update algorithm, which have different tradeoffs between speed of convergence and latency.



*(a) Timed event graph*

| **Style** | **Waiting primitive (§4)** |
|---|---|
| Bold | ProducerAcquire |
| Dashed | ConsumerAcquire |
| Solid | ProducerSend |
| Dotted | Computation |

*(b) Types of edge*

**Fig. 5.** Example timed event graph used by the critical cycle algorithm

### 5.1 Cycle Detection Algorithms

**Critical cycle algorithm** The *critical cycle* algorithm [14–16] solves the cycle detection problem for homogeneous Synchronous Data Flow (SDF) [17] with constant computation times and communications latencies. In homogeneous SDF, every time a producer or consumer fires, it pushes or pops a single buffer on each stream. All tasks therefore fire at the same rate. The algorithm can be extended to SDF, where each producer or consumer pushes or pops any fixed number of buffers, but it requires expanding the graph, which can make it much bigger [18].

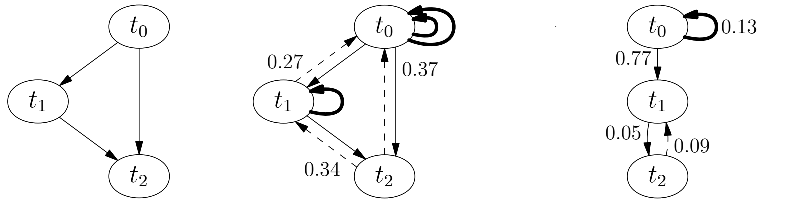
Figure 5(a) shows how producer-consumer, assuming a single buffer at each end, is represented by this algorithm. Each vertex is the return from a communications primitive. The edges are distinguished, for the diagram but not the algorithm, using the convention in subfigure (b), which refers to the primitives in Section 4. Each edge has *weight*, which is its fixed computation time or communications latency, and *height*, which is the fixed difference between the firing number, which counts the number of times a task has fired, at its two ends.

For example, at the producer side, the dotted line from ProducerAcquire to ProducerSend, of weight 448 and height 0, represents computation inside a single iteration. The solid line in the reverse direction, of weight 13 and height 1, is because the producer cannot reuse its single buffer in the current firing until the previous DMA has completed.

Throughput is constrained by the *critical cycle*, which is a cycle with maximum ratio of total weight divided by total height. There are several algorithms to find such a cycle, many based on Karp's Theorem [19], in time $O(|S|^2|T|)$ or so [15], using the terminology of Section 4.

**Baseline Algorithm** Our *baseline* algorithm is more general, because it supports variable data rates, computation times, and communication latencies. It finds the bottleneck by analysing wait times in a real execution or simulation.

Figure 6 shows how the stream program and wait times are represented by the algorithm. Subfigure (a) is an example stream graph with three tasks in a triangle. Subfigure (b) is the *wait-for* graph, which has the same three edges per stream as the timed event graph. Following convention for wait-for graphs, the arrows point in the opposite direction, *from* the waiting task. The weight of an edge is the proportion of the total time that the task at the initial vertex, or tail, spent waiting in its communications primitive.



*(a) Program* *(b) Wait-for graph* *(c)* $(t_0)$ *has zero strength*

**Fig. 6.** Example weighted wait-for graphs

As for the critical cycle algorithm, performance is constrained by dependence cycles in the wait-for graph. We will use two bounds, one local and one global, on the maximum increase in performance from *relaxing* a cycle; i.e. increasing buffering on one of the streams in the cycle that gets full.

Consider the potential benefit from relaxing cycle $C_1 = (t_0\, t_2\, t_1)$. This can only be done by increasing buffering on the stream from $t_0$ to $t_2$. Since $t_1$ waits for 27% of the time, during the ConsumerAcquire primitive in this cycle, we could reduce the execution time of $t_1$ by *at most* 27%, before the cycle disappears. Since all tasks execute for nearly the same amount of wallclock time, any change in throughput will cause all vertices to have their *total* waiting time, not just on the edges of this cycle, reduced by the same amount. It is therefore likely that the edge in the cycle that disappears first is its weakest edge.

The local bound is the *weight* of cycle $C$, denoted $w(C)$, which is the minimum weight of its edges. If there is no cycle with non-zero weight, then utilisation is already 100%. This is because every directed *acyclic* graph has a vertex with no outgoing edge, which corresponds to a task that never has to wait.

Figure 6(c) is the motivation for the global bound. The maximum weight cycle is the loop on $t_0$, of weight 0.13, which we will call $C_2$. A moment's reflection, however, shows that $C_2$ cannot really be a bottleneck since neither $t_1$ nor $t_2$ ever wait for $t_0$, even indirectly. If we reduced the time $t_0$ spent waiting on this loop, it cannot make $t_1$ or $t_2$ go any faster. Since throughput would be unchanged, $t_0$

must spend the same total amount of time waiting, so the waiting time would move from ProducerAcquire to ProducerSend (see Figure 5(b)).

The global bound is the *strength* of the cycle, denoted $s(C)$, which is the lowest value of the maximum flow *through a single path* to the cycle, starting from any other vertex. Since there is no path at all from $t_1$ to $C_2$ in Figure 6, the cycle has zero strength: $s(C_2) = 0$. In contrast, the cycle $(t_1\, t_2)$ has strength 0.77, because this is the weight of the only path from the only other vertex, $t_0$. Increasing the performance of $t_1$ and $t_2$ by any means could reduce execution time of the program as a whole by 77%. This cycle is the bottleneck, and it has weight 0.05. The requirement that flow be through a single path makes little difference in practice, but it reduces considerably the algorithmic complexity.

It is possible for the wait-for graph to be disconnected; e.g. when tasks wait for each other only through bus contention. This happens rarely, but it causes all strengths to be zero. Therefore, when all strengths are zero but the utilisation is below some threshold (currently 100%), the strengths are ignored. Since it almost never happens, there is little reason to be more sophisticated.

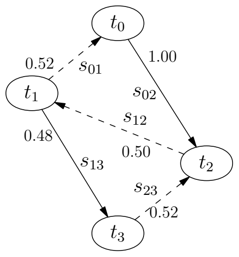
We first calculate the strength of each vertex by computing the *all-pairs bottleneck paths* [20]. This finds, for every pair of vertices, the value of the maximum flow through a single path from the first vertex to the second. It is solved using a variant of Dijkstra's algorithm, running Dijkstra for each vertex to find the maximum flow paths into it. The strength of that vertex is given by the path with the lowest flow. The total execution time is $O(|S||T| + |T|^2 log|T|)$, using a Fibonacci heap [21, 22], with the terminology of Section 4.

The algorithm finds a cycle with the maximum value of the minimum of the local and global bounds. It is straightforward to show that we can take account of both simply by replacing the weight of every edge $e = (a, b)$ by a new weight, $w'(e) = \min(w(e), s(a))$. A maximum weight cycle, according to $w'$, can be found in time $O(|S| \log |S|)$, where $S$ is the set of streams. To find out whether there is a cycle of weight $\geq W$, for some $W$, just check whether there is any cycle if you ignore all edges of weight $< W$. This can be done in time $O(|S|)$ by attempting to perform a topological sort. To find a maximum weight cycle, first sort the edge weights, and perturb them so that no two are exactly the same. Then use bisection on the sorted edge weights.

The baseline algorithm uses data that is easy to obtain, and is usually quite effective, but it has one limitation. Since each task is represented by a single vertex, it cannot "see" what is happening inside them.

Figure 7(a) shows an example where the baseline algorithm makes a bad decision. The maximum weight cycle is $(t_1\, t_0\, t_2)$, which has weight 0.50. Whether or not this is a bottleneck depends on the internal behaviour of tasks $t_1$ and $t_2$. The order of operations per firing of task $t_1$ is shown in subfigure (b). If we also know that task $t_1$ *always* waits in step 5, then reducing the waiting time in step 1 will simply result in a longer waiting time in step 5. It can never advance the push in step 6, so the critical cycle cannot be $(t_1\, t_0\, t_2)$.

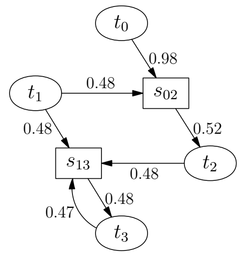
**Token Algorithm** The *token* algorithm addresses this problem by tracking dependencies through tasks. This is somewhat similar to causal chains [23], except



(a) *Wait-for graph*

| Primitive | Wait time |
|---|---|
| 1. ConsumerAcquire($s_{01}$, 0) | 0.52 |
| 2. ProducerAcquire($s_{13}$, 0) | *n/a* |
| 3. ProducerAcquire($s_{12}$, 0) | *n/a* |
| 4. ConsumerDiscard($s_{01}$) | *n/a* |
| 5. ProducerSend($s_{13}$) | 0.48 |
| 6. ProducerSend($s_{12}$) | *n/a* |

(b) *Order of primitives in* $t_1$



(c) *Indirect wait-for graph*

**Fig. 7.** Example where baseline fails

that the aim is to resolve performance bottlenecks rather than artificial deadlocks. Their algorithm fixes a deadlock after it happens, when all tasks have got stuck, but we cannot expect all tasks in a cycle to ever be waiting simultaneously.

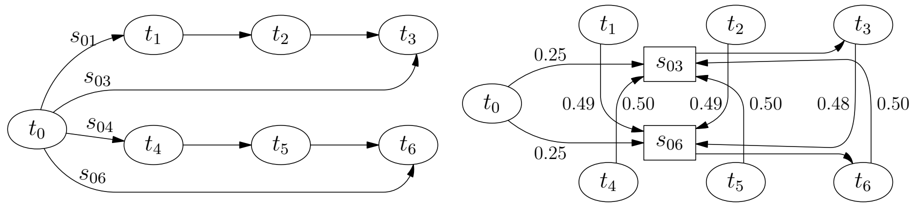
During the simulation, or at runtime in a dynamic scheme, each task $t$ has a *current token*, $S_t$, which is the stream that most recently made $t$ wait, directly or indirectly, because it got full. It has a *current waiting time*, $W_t$, which measures how much the task has already had to wait, so that only increases in waiting times are charged to streams. It also has a *waiting vector*, $(V_t)_s$, which gives the total waiting time for each stream in the whole program. Each consumer buffer $c$ has a *current token*, $S_c$, and *current waiting time*, $W_c$, which together record the producer's problem at the time the block in that buffer was sent.

When task $p$ blocks for time $\tau$ because output stream $s$ is full, it sets $S_p \leftarrow s$ and increases both $W_p$ and $V_p[s]$ by $\tau$. When task $p$ sends a block using buffer $c$ on output stream $s$, it records a copy of its current state: $S_c \leftarrow S_p$ and $W_c \leftarrow W_p$. When a task $q$ blocks for time $\tau$ because input stream $s$ is empty, it also, after the data arrives, reads $S_c$ and $W_c$, from the consumer buffer $c$ containing the end of the data. It then updates its current token $S_q \leftarrow S_c$ to indicate that it had to wait, indirectly, for whichever stream the producer had to wait for, and calculates the increase in current waiting time $\Delta W_q \leftarrow \min(\tau, W_c - W_q)$, which can be either positive or negative. If it is positive, then $V_q[S_q]$ is increased by $\Delta W_q$. In either case, the current waiting time is then updated using $W_q \leftarrow W_q + \Delta W_q$.

The waiting vectors are used to construct an *indirect wait-for graph*, as shown in Figure 7(c). If $V_t[s] > 0$, there is an edge from task $t$ to stream $s$ with weight $V_t[s]/L$, where $L$ is the total execution time of the run, in the same units. Each stream $s$ also produces an edge from $s$ to its consumer $q$. The weight of this edge is $s(q)$, the *strength* of $q$, as defined for the baseline algorithm.

This is effectively viewing each stream as an actor in its own right, which is always blocked waiting for the consumer to discard its data. This is the most convenient place to take account of the strengths, which are still relevant by the same argument as before. The token algorithm finds the maximum weight cycle in the same way as the baseline algorithm.

Figure 8 shows a second example which clarifies the need for the cycle-based algorithm outlined above. In the stream program of Figure 8(a), task $t_0$ pushes the outputs in the cyclic order ($s_{01}\, s_{03}\, s_{04}\, s_{06}$), waiting only in ProducerSend for streams $s_{03}$ and $s_{06}$ due to their longer latency.



*(a) Stream graph for bichain4* *(b) Indirect wait-for graph*

**Fig. 8.** Token algorithm: bichain4 example

When it pushes on stream $s_{04}$ of the right branch, the most recent wait was due to stream $s_{03}$ being full, so it sends the token for $s_{03}$. Similarly, it sends the token for stream $s_{06}$ to stream $s_{01}$ of the left branch. The indirect wait-for graph is shown in Figure 8(b), with cycle ($t_3\, s_{06}\, t_6\, s_{03}$) going through both streams.

## 5.2 Buffer Size Update Algorithms

The cycle detection algorithm returns a set of edges in the wait-for graph that cause a bottleneck cycle by becoming full. Relaxing the cycle involves increasing memory on one or more of these edges. The purpose of the buffer size update algorithm is to determine *which* edges to enlarge, and by *how many* buffers.

Our simplest algorithm is *miserly*, meaning that it starts at the minimum number of buffers, mentioned in Section 4, and each iteration increases the allocation of a single buffer by one. The other algorithms speculatively assign spare memory, and only take it away if it is needed elsewhere. For all these algorithms, each stream $s$ *demands* some number $d_s$ of buffers, as for the miserly algorithm, and *requests* another $r_s$ to be granted out of unused memory, if there is any. When there is not enough memory to grant all requests within some memory, we used the following algorithm. The total request in bytes is $R = \sum r_s b_c(s)$, where $b_c(s)$ is the size in bytes of a single consumer buffer for stream $s$. If $M$ bytes are left after granting all demands, so $R > M$, then each stream is initially granted $\lfloor r_j M/R \rfloor$ extra buffers, then possibly one more, if it fits.

In our first alternative, *double*, each edge requests an extra buffer if it is currently allocated only one. In our second alternative, *exponential*, the request is for some multiple, $f - 1$, of the number of buffers demanded. We still use a greedy update algorithm, so that when the number of buffers is increased, the edge demands, on the next iteration, one more buffer than it was given in total last time. We used $f = 2$, so an edge will demand $2^k - 1$ buffers, and request an equal number, for $k = 1, 2, \cdots$, until it is given fewer buffers than it wants.

The third alternative, *level*, uses the *top level*, the length of the longest path from a source node, and *bottom level*, the length of the longest path to a sink node. The algorithm the same as *exponential*, except that the request is the maximum of a) $f - 1$ times the number of buffers demanded, b) twice the difference in top level, and c) twice the difference in bottom level. This tries to give a high initial allocation to streams that cross a high latency.

## 6 Evaluation

We used the StreamIt 2.1.1 benchmarks [24], random graphs, and sixteen examples, including *chain8*, *producer-consumer*, *bad-baseline*, and *bichain4*. For the StreamIt benchmarks, we used the program graph, work estimates and communications rates generated by the StreamIt compiler, and used our algorithm [25] to produce partitions for an IBM QS20 blade, which has two Cell BEs.

**Buffer size update** The first three rows of Figure 9 compare the buffer update algorithms from Section 5.2. These plots also contain results for Basten and Hoogerbrugge (B&H) [23] and modified StreamRoller [3], which will be discussed in Section 7. The left column shows as a function of the iteration number, the utilisation, which is proportional to throughput, as remarked at the end of Section 2. The right column shows the tradeoff between latency and utilisation. Any points that cannot be Pareto optimal, because they are beaten on both utilisation and latency by some point to the top-left, have been removed.

The first row is for random stochastic graphs with 32 tasks and 50 streams. The graphs are connected and acyclic, but otherwise unstructured. The computation time of each task is normally distributed with a random mean and variance (clamped above zero). Notice that B&H has poor performance and, since it increases buffering where it isn't necessary, high latency.

We found the upper bound on utilisation using an exhaustive search over all allocations of the buffers on the processor, $p$, whose memory bound caused the level algorithm to terminate. All other queues on other processors were set to their maximum possible size, assuming that all other queues in the same memory had their minimum size. Since this tends to allow a task near the beginning of the stream graph to work flat out filling downstream buffers, the steady state utilisation would be known only after many firings. Instead, we took the utilisation of the task on $p$, and scaled by the ratio of the long-term processing times of the most heavily loaded processor and of $p$.

The second row shows the StreamIt 2.1.1 benchmarks, with an unroll factor of 100. The third row shows the *stochastic* StreamIt benchmarks, which have normally-distributed computation times, and are intended to show how the algorithms fare for realistic program graphs.

The left column shows that the level algorithm always provides the fastest convergence. The modified StreamRoller algorithm is similar to the first iteration of the level algorithm, and B&H is considerably worse. The level heuristic initial allocation is within 15% of the upper bound on optimal performance, and is increased to within 3% of optimal after four iterations.

**Cycle detection** We evaluate the cycle detection algorithms only, using greedy buffer update without memory constraints. When task execution times and communications rates are constant, and bus contention is negligible, the critical cycle algorithm of Section 5.1 is optimal. The last row of Figure 9 shows the utilisation and latency for an average of six random graphs with stochastic computation times. The poor performance of the critical cycle algorithm (about 60% utilisation), is because it is unable to detect cycles that arise from execution time variability. The baseline and token algorithms achieve similar performance, although the token algorithm achieves slightly lower latency.

We also evaluated the cycle detection algorithms when there is high bus utilisation, but for space reasons did not include the graph. The critical cycle algorithm cannot model increased communication latency due to contention [26, §E.5]. For a benchmark with a single producer task connected to two consumers, and bus usage close to 100%, the critical cycle algorithm achieves about 70% utilisation. The baseline and token algorithms measure waiting times directly, and consistently achieve 100% utilisation.

## 7 Related Work

Basten and Hoogerbrugge (B&H) [23] is the only other work that also targets unstructured graphs with variable multiplicities and computation times. Their algorithm sets each FIFO buffer size to be proportional to the amount of data streaming through it. This gives a relative size for each buffer, but it is not motivated by the underlying problems discussed in Section 2, and has poor performance in Figure 9. We interpreted B&H to mean double buffering on the producer side, with all the remaining memory allocated to consumer buffers, rounding the number of buffers *up* to an integer. If rounding up causes the buffer allocation to not fit, we reduced the target memory use until it did fit. The *chain8* example in Figure 1 shows the problem with this heuristic. If all data rates are the same and there is enough memory on $t_n$ for ten buffers, Basten and Hoogerbrugge allocates five buffers to each stream for 70% utilisation, while our heuristic allocates eight to $(t_1, t_n)$ and two to $(t_{n-1}, t_n)$ for 100% utilisation.

The SDF tool [27] uses an exhaustive search to find all Pareto-optimal buffer allocations for an SDF graph. It requires exponentially many steps, and only supports constant computation times and data rates. For an $n$-way split or join where each stream needs $b$ buffers, their algorithm requires $n^b$ steps, while our level algorithm requires $O(n \log_2 b)$ steps to find a single solution.

StreamRoller [3] performs buffer allocation as part of software pipelining, but it is restricted to graphs with fixed multiplicities and computation times. The algorithm is similar to the first iteration of the *level* algorithm, in that the number of buffers allocated to a stream is always one plus the difference in pipeline stage. The *chain8* example in Section 2 shows that this is conservative, even when there is no variability. Hence the StreamRoller algorithm can require more memory than necessary; if there is insufficient memory, it fails.

Due to the unrolling factor we used, StreamRoller failed on at least one benchmark for all of the graphs in Figure 9. This is true even for the StreamIt benchmarks, for which our algorithm achieves 100% utilisation on at least one

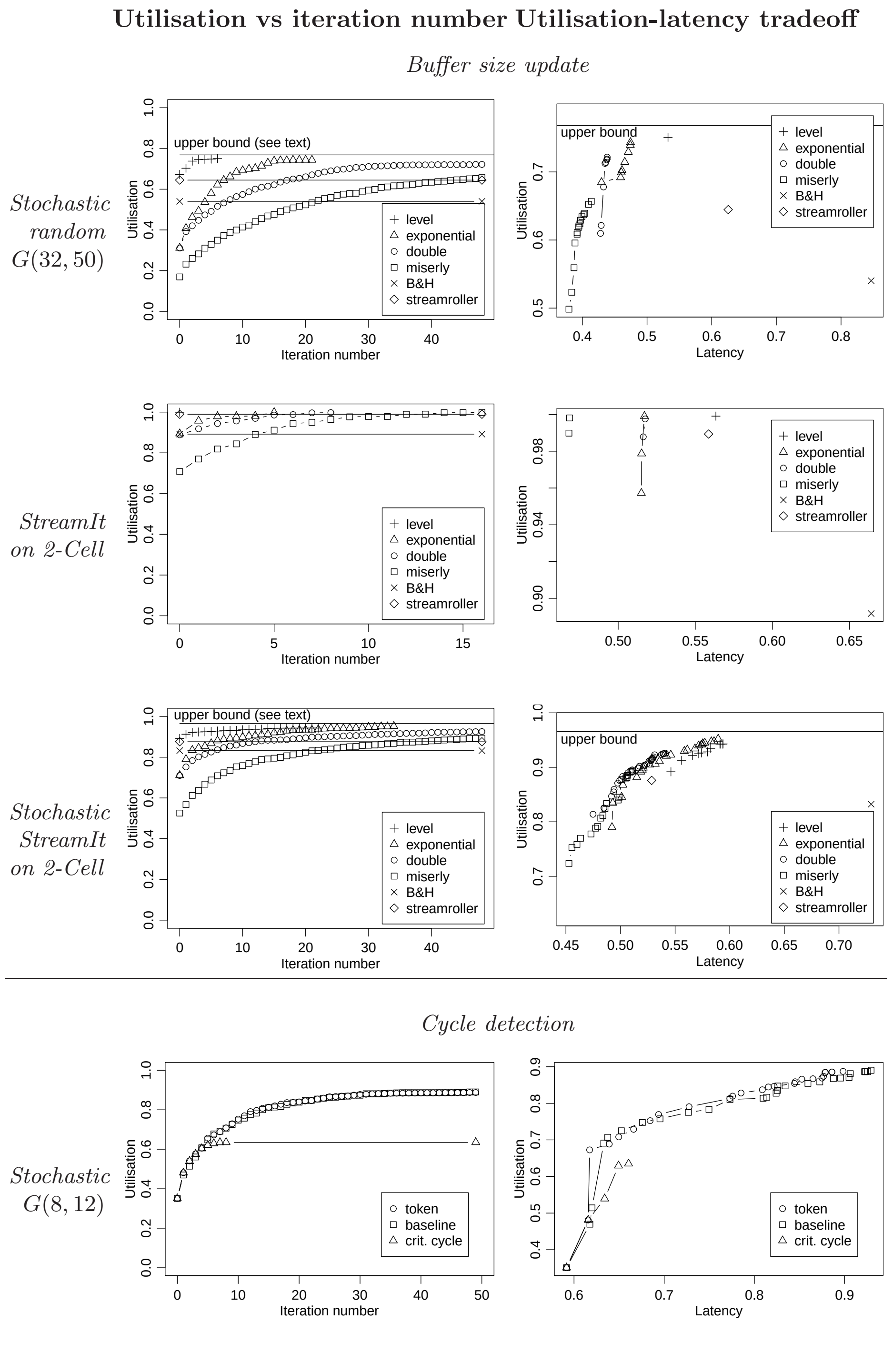


**Fig. 9.** Comparison of the buffer size update and cycle detection algorithms

processor. We modified StreamRoller to use our arbitration scheme described in Subsection 5.2, and obtained the results shown in Figure 9. Even with this modification, however, our iterative algorithm has about 13% higher performance for the stochastic random graphs and stochastic StreamIt benchmarks.

The SPIR compiler [4] extends StreamRoller to find a partition and software pipeline subject to memory and latency constraints. Unlike our approach, computation times and communication rates are constant. As for StreamRoller, the number of buffers allocated to a stream is one plus the difference in pipeline stage. Since the problem cannot be solved exactly using ILP, it is a heuristic which uses two passes of the commercial CPLEX ILP solver. Our algorithm could be used to improve the buffer allocation of a partition produced by SPIR.

## 8 Conclusions

In this paper, we presented a feedback-directed algorithm to allocate memory for communications buffers in a statically-allocated stream program. The algorithm achieves close to optimal performance, even when StreamRoller fails due to insufficient memory. It achieves significantly higher performance and lower latency than the previous fully general algorithm, by Basten and Hoogerbrugge.

### Acknowledgements

The researchers at BSC-UPC were supported by the Spanish Ministry of Science and Innovation (contract no. TIN2007-60625), 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.

## References

1. Olukotun, K., Hammond, L.: The future of microprocessors. Queue **3**(7) (2005) 26–29
2. Pham, D., Behnen, E., Bolliger, M., Hofstee, H., et al.: The design methodology and implementation of a first-generation Cell processor: a multi-core SoC. In: Custom Integrated Circuits Conference, 2005. (2005) 45–49
3. Kudlur, M., Mahlke, S.: Orchestrating the execution of stream programs on multicore platforms. Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation (2008) 114–124
4. Choi, Y., Lin, Y., Chong, N., Mahlke, S., Mudge, T.: Stream Compilation for Real-Time Embedded Multicore Systems. In: Proceedings of the 2009 International Symposium on Code Generation and Optimization-Volume 00. (2009) 210–220
5. IST-034869, A.: http://www.hitech-projects.com/euprojects/ACOTES/ Advanced Compiler Technologies for Embedded Streaming.
6. ACOTES: IST ACOTES Project Deliverable D2.2 Report on Streaming Programming Model and Abstract Streaming Machine Description Final Version. (2008)

7. Becchi, M., Crowley, P.: Dynamic thread assignment on heterogeneous multiprocessor architectures. In: Proceedings of the 3rd conference on Computing frontiers, ACM New York, NY, USA (2006) 29–40
8. Hofstee, H.P.: Power efficient processor architecture and the cell processor, Los Alamitos, CA, USA, IEEE Computer Society (2005) 258–262
9. Parks, T.: Bounded scheduling of process networks. PhD thesis, University of California (1995)
10. Buck, J.: Scheduling dynamic dataflow graphs with bounded memory using the token flow model. PhD thesis, University of California (1993)
11. Geilen, M., Basten, T.: Requirements on the execution of Kahn process networks. Lecture Notes in Computer Science (2003) 319–334
12. van der Wolf, P., de Kock, E., Henriksson, T., Kruijtzer, W., Essink, G.: Design and programming of embedded multiprocessors: an interface-centric approach. Proceedings of the 2nd international conference on Hardware/software codesign and system synthesis (2004) 206–217
13. Carpenter, P.M., Ramirez, A., Ayguade, E.: The Abstract Streaming Machine: Compile-Time Performance Modelling of Stream Programs on Heterogeneous Multiprocessors. In: SAMOS Workshop 2009, Springer 12–23
14. Ito, K., Parhi, K.: Determining the minimum iteration period of an algorithm. The Journal of VLSI Signal Processing **11**(3) (1995) 229–244
15. Dasdan, A., Gupta, R.: Faster maximum and minimum mean cycle algorithms for system-performance analysis. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **17**(10) (1998) 889–899
16. Govindarajan, R., Gao, G.: A novel framework for multi-rate scheduling in DSP applications. In: International Conference on Application-Specific Array Processors. (1993) 77–88
17. Lee, E., Messerschmitt, D.: Synchronous data flow. Proceedings of the IEEE **75**(9) (1987) 1235–1245
18. Lee, E.A.: A coupled hardware and software architecture for programmable digital signal processors (synchronous data flow). PhD thesis (1986)
19. Karp, R.: A characterization of the minimum cycle mean in a digraph. Discrete mathematics **23**(3) (1978) 309–311
20. Pollack, M.: The maximum capacity through a network. Operations Research (1960) 733–736
21. Fredman, M., Tarjan, R.: Fibonacci heaps and their uses in improved network optimization algorithms. Journal of the ACM (JACM) **34**(3) (1987) 596–615
22. Vassilevska, V., Williams, R., Yuster, R.: All-pairs bottleneck paths for general graphs in truly sub-cubic time. In: Proceedings of the thirty-ninth annual ACM symposium on Theory of computing, ACM New York (2007) 585–589
23. Basten, T., Hoogerbrugge, J.: Efficient execution of process networks. Communicating Process Architectures (2001)
24. Gordon, M., Thies, W., Amarasinghe, S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. ASPLOS (2006) 151–162
25. Carpenter, P.M., Ramirez, A., Ayguade, E.: Mapping Stream Programs onto Heterogeneous Multiprocessor Systems. In: CASES '09, October 11–16, 2009
26. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach. Fourth edn. Morgan Kaufmann (2007)
27. Stuijk, S., Geilen, M., Basten, T.: Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In: Proceedings of the 43rd annual conference on Design automation. (2006) 899–904