The CDAG: A Data Structure for Automatic Parallelization for a Multithreaded Architecture

Bernd Klauer, Frank Eschmann, Ronald Moore, Klaus Waldschmidt*
J.W.Goethe-University, Technical Computer Sc. Dep., Box 11 19 32, D-60054 Frankfurt, Germany
{klauer|eschmann|waldsch}@ti.informatik.uni-frankfurt.de
Published in: The proceedings of the 10th Euromicro Workshop on
Parallel, Distributed and Network-based Processing, Canary Islands, Spain, January 2002.

Abstract

Despite the explosive new interest in Distributed Computing, bringing software — particularly legacy software — to parallel platforms remains a daunting task. The **S**elf **D**istributing **A**ssociative **ARC**hitecture *(SDAARC) takes a two-fold approach to this problem. Seemingly sequential programs are first translated into a population of migratory* threads and containers *by the compiler, and then allowed to migrate to minimize communication while maximizing parallelism by a run time environment.*

However, previous compilers for multithreaded architectures such as SDAARC did not permit the full range of control flow complexity found in programming languages such as C. Thus, we propose a new data structure, and present algorithms for its construction, which extends the familiar concepts of control flow and dataflow graphs to conveniently represent the activities required of an automatically generated thread.

1 Introduction

The last few years have seen a renaissance in interest in Distributed Computing, typified by the interest shown for project such as Seti@Home [4][5]. However, while much effort is being invested in the development of new platforms for distributed computing, less progress is being made at lowering the barriers the make porting programs to these platforms difficult. Programming explicitly parallel programs is and remains difficult. The difficulty of programming parallel programs stems from two main sources:

1. There is still no single unifying model for parallel computation which can play the role that the so-called von Neumann model plays for sequential computation. Further, with the renewed proliferation of new platforms, the emergence of an industry consensus does not seem likely any time soon.
2. Programming parallel programs is *inherently* more difficult than programming sequential programs, simply because the number of dimensions in the problem space is greater: instructions much be *distributed* across space and not only time and, in the absence of a true shared memory, data must be *distributed* across multiple address spaces. Further, these distributions must seek to find a compromise between maximizing load balance and minimizing communication costs —- two goals that quite often pull the distribution process in diametrically opposite directions.

The S*elf* D*istributing* A*ssociative* ARC*hitecture* (SDAARC) was proposed in [9][8][10][6][11] to address these issues. The goal of the project was (and is) to find a way to combine static (compile time) analysis of legacy software — programs written in conventional languages such as C and Fortran, without explicit parallel constructs — with a dynamic (run time) environment to tackle the *distribution problem*, and to do so in a way that could be applied across various scales and hardware implementations. The approach combines and extends ideas from multithreaded architectures with those from **C***ache* **O***nly* **M***emory* **A***rchitecture* (COMA) to target a system which can convert legacy software into a migratory population of *threads and containers*, which can then react dynamically and adaptively to unpredictable conditions as they emerge at run time.

COMAs provide a role model that show how the local memories in a distributed computing platform can be treated conceptually as (large, slow) *caches* — memories that hold only the data their processors need, and which *evict* data blocks which are no longer in use [3][13]. Where a conventional cache evicts data blocks to a backing store, the local memories in a COMA — called *attraction memories* (AMs), in order to distinguish them from conventional

*Parts of this work have been supported by the Deutsche Forschungsgemeinschaft (DFG).

caches — evict data blocks by necessity to *other* AMs. Thus, data blocks in a COMA are *migratory* entities.

COMAs move the responsibility for data distribution away from the programmer and/or the compiler to a run time system, but they still presume that the *computations* performed on the data have been explicitly distributed across the processors. The idea behind SDAARC was to turn computation into a second population of migratory entities.

For these entities we choose *microthreads*, inspired by the work done in [12]. Microthreads are the smallest computational units in SDAARC, and represent computations which contain no useful parallelism and which can thus be run sequentially. Microthread-boundaries are drawn by the compiler at control structures and at global data read and write operations, as described in more detail in [7].

The basic concept behind SDAARC is very simple: An instance of a microthread is represented at run time by a data-structure called a *microframe*.[1] There is a one-to-one mapping between microframes and microthreads which need (eventually) to be scheduled. A microframe stores (among other things) the arguments of its microthread. When the microframe is *full*, the microthread is *ready* to execute.

The microframes are stored in a special partition of the Distributed Shared Memory, but migrate in much the same manner as conventional COMA data blocks. Each site in the SDAARC Ensemble maintains its own scheduling queue. When a microframe becomes full, its microthread is placed in the local scheduling queue. Further migration between one scheduling queue and another to perform load balancing is possible, based on the principle of *work stealing* [2][11]. Thus, *frame migration implies thread scheduling*.

SDAARC's Compiler thus needs to meet the following requirements:

- It needs to accept arbitrary programs in legacy programming languages, in particular, in C;
- It needs to dissect these programs into sets of microthreads;
- It needs to generate code for each microthread — in particular, it must generate code not only for the computations found in the original source code, but also for the mechanics unique to microthreads, such as communication and the dynamic creation of *other* microthreads.

Early compilers developed for SDAARC met the last two requirements, but were based on *syntax-tree* analysis, and were restricted to well-structured subsets of the full syntax of C. As we extended this subset, we eventually hit upon limitations that required us to rethink how we should represent and manipulate legacy software. Even without allowing the much criticized "`goto`" statement, the syntax of C's "`switch`" statement allows almost arbitrarily complex control flow to enter into even seemingly simple programs.

We thus decided to work from control flow graphs instead of syntax trees. However, conventional control flow graphs, and their cousins, conventional dataflow graphs, proved insufficient for the requirements of code generation for SDAARC, since they did not sufficiently address the problems associated with the dynamic allocation of *new microthreads*. Thus, we devised a data structure we call a CDAG — a Control Dataflow Allocation Graph.

Further, we soon saw that there were two styles of algorithms to choose for when constructing CDAGs: *all-at-once* or *step-by-step*. At first we hoped to derive simple mathematical definitions for the correct placement of all of the edges in the CDAG. We quickly realized however that more than one correct set of edges exist for anything but artificially small programs. Further, we saw that it is relatively easy to generate a semantically correct, but completely sequential CDAG directly from the control flow graph — and then to define a set of optimizations that introduce parallelism intro the graph is small steps. We have thus chosen to pursue this *step-by-step* approach in order to leave room for future development of new optimizations.

In short, we see the CDAG as a compact formalism which provides a clear and succinct basis for the extraction, representation and optimization of the operations which must be performed in a multithreaded architecture — the communication of results between threads, and the selection and allocation of new threads at run time.

2 The Compiler Techniques

The tasks to be performed by a thread are the following:

1. load the thread's arguments out of the thread's microframe,
2. compute the thread's results,
3. make decisions on the executability of other threads in the future,
4. allocate microframes for the new threads (decided on in the step 3,
5. send the results (calculated in step 2) to their recipients.

At compile time the code generator of the compiler has to generate code for each of these phases. The code generation of phase 2 can be performed using standard compiler technology. For the proper generation of the other phases, we introduce a data structure called **C***ontrol* **D***ataflow* **A***llocation* **G***raph*(CDAG). The CDAG consists of:

[1] Similar to the "framelets" introduced in [1].

- Nodes representing the microthreads,
- Edges representing the control flow,
- Edges representing the dataflow,
- Edges representing the allocation flow.

The nodes represent the microthreads. In a preliminary step, the source program is partitioned into threads at boundaries dictated by control structures and global storage usage. See [7].

Each node has data input and output ports for the dataflow edges. If the thread makes decisions then its node also has *cases*. The total of cases depends on the total of decisions that can be taken. Nodes without decisions have exactly one case. A simple `if-then-else` construct produces 2 cases. A `switch{case 1: ... case n:}` construct produces n cases. **The control flow edges** represent the control flow between the Microthreads. They are used only by the compiler for traversing the graph efficiently, and have no counterpart at run time. **The dataflow edges** represent the data dependencies and the communication links between the microthreads. They are used by the code generator to produce code for the phases 1 and 5. **The allocation edges** represent the frames that have to be generated by a thread. Allocation edges as well as dataflow edges depend on the cases of a node. As more than only one frame can be allocated from one thread, multiple threads can later be selected by the scheduler to be executed on different machines. Thus, the allocation edges are the key for the parallelization process.

A variety of analysis steps will be shown in the following sections to derive the dataflow and the allocation flow from the syntax tree. As we will see later more than one placement of allocation edges is possible. Thus, the generation of allocation edges is subject to optimization.We distinguish 3 general allocation policies:

- naive allocation
- earliest allocation
- just-in-time allocation

Naive Allocation: The *naive* allocation directly follows the control flow. Each thread allocates exactly one successor microframe. The naive allocation produces valid code which is however completely devoid of parallelism. The result of this policy is interesting as a starting point for the optimization procedures and to produce sequential code for run-time comparisons.

Early Allocation: A Microframe can be allocated as soon as we now that its corresponding code will executed at least once. The advantage is that the time between the generation of a frame and the execution of the corresponding thread is long. This provides time for the scheduling mechanism to migrate the frame to a *underutilized* site. On the other hand the machine can be flooded with Threads waiting on their arguments.

Just-In-Time Allocation: Frames must be available as soon as they are the receiver of data transmissions from other threads. So the latest allocation would be performed by the first transmitter of data. The advantage of this policy is that the scheduler has to care of a smaller set of frames.

Finding the probably best policy among the last two strategies — or in between — will be subject of future work. In the following sections we discuss the generation and optimization of the CDAGs and we show how they are used in the code generation process.

2.1 Definitions

A microthread contains only code which has $\mathcal{O}(1)$ run time complexity. To indicate that a symbol m represents a microthread we use the following denotation:

$$\dot{m}$$

The control flow case determines which of the successor microthreads (with respect to the control flow) will be executed. For example an `if-then-else` construct has 2 control flow cases, one for the `then`-branch and one for the `else`-branch. A control flow between two microthreads $\dot{a}$ and $\dot{b}$ taking the control flow case i is denoted by:

$$\dot{a}_i \xrightarrow{C} \dot{b}$$

We now introduce *routing threads*. They are special nodes without any computation. They are just transceivers for data. Their special feature is that they can hold data until the receiver of the data has been determined (at run-time). A symbol representing a routing thread r is denoted by

$$\hat{r}$$

A dataflow edge between two microthreads $\dot{a}$ and $\dot{b}$ carrying the argument x in the control flow case i is denoted by:

$$\dot{a}_i \xrightarrow[x]{D} \hat{b}$$

Dataflow edges can also exist between routing threads or between routing threads and microthreads.

The allocation of microframe $\dot{b}$ by microthread $\dot{a}$ when taking control flow case i is denoted by:

$$\dot{a}_i \xrightarrow{A} \dot{b}$$

Routing frames may also be allocated, but only microthreads can allocate frames (because routing threads don't execute any code).

Figure 1 shows a graphical representation of a CDAG. The graph consists primarily of a `if` ($\dot{b}$) `then` ($\dot{c}$) `else` ($\dot{d}$) instruction. Routing frame $\hat{b}$ transports a variable to $\dot{c}$ or $\dot{d}$, depending on the control flow decision made by microthread $\dot{b}$.

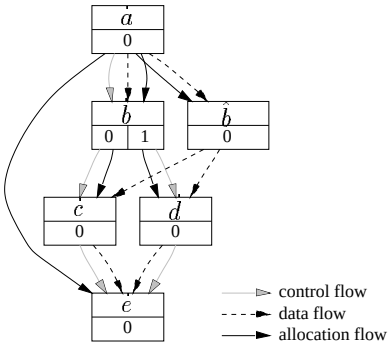


Figure 1. Control Data Allocation Graph (CDAG)

2.2 CDAG generation and optimization

To show how our compiler generates CDAGs and how it optimizes the CDAGs for parallelization and for the code generation, we use the following simple example C-program:

```
main()
{
           unsigned int a,b,c;
           a=10;
           b=20;
           goto live_code;
dead_code: a=10*a;
live_code: c=20*a+10*b;
           if(c>10)
           {
              c/=2;
              c-=b;
           } else
           {
              c*=2;
              c+=b;
           }
           return a+b;
}
```

After scanning and parsing the original C-code the SUIF Compiler produces a syntax tree. After deriving the code partitions we produce the control flow graph in figure 2.

In the first optimization pass we eliminate dead code (code, that can never be reached; in the example Micro8), using the implication

$$\left(\nexists \dot{a}, i : \dot{a}_i \xrightarrow{C} \dot{b}\right) \Rightarrow \dot{b} \text{ is dead code}$$

and poor code (code thalertech.text does *nothing*, i. e. microthreads that neither change nor create any arguments needed by other microthreads).

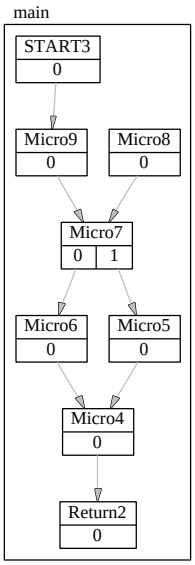


Figure 2. A simple control flow graph

We then add allocation edges with a *naive* procedure. Each pair of microthreads that are connected by a control flow edge will be connected by a allocation edge (see fig. 3):

$$\dot{a}_i \xrightarrow{C} \dot{b} \Rightarrow \text{create } \dot{a}_i \xrightarrow{A} \dot{b}$$

(a) before

(b) after

Figure 3. naive generation of allocation edges

The naive allocation graph will later be subject to optimization. The result is the graph in fig. 4.

A SDAARC program generated using this graph would run strictly sequentially. The next step on our way to parallelism is that we move the starting points of all allocation edges to the predecessor who decides if the target node will be allocated (and executed). Such nodes are branch nodes (a node with control flow fanout > 1) and start nodes. The `early_allocation` procedure consists of three steps:

1. All allocation edges in sequential chains in the graph are moved to the node before the sequence. Such nodes are branch nodes, merge nodes (nodes with control flow fanin ¿ 1) or start nodes. Formally, if a microframe $\dot{b}$ isn't a branch

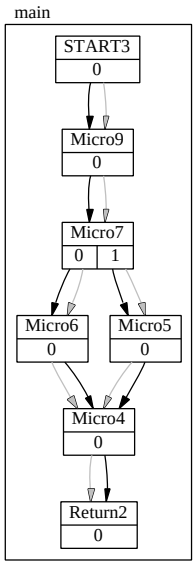


Figure 4. A simple control/allocation graph

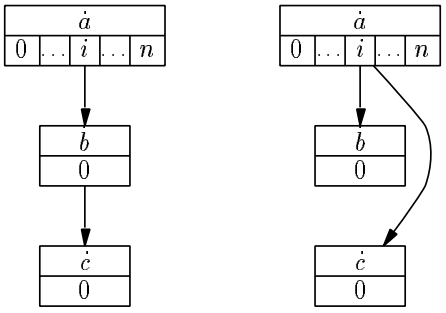
or a merge-node, the allocation edge

$$\dot{b}_0 \xrightarrow{A} \dot{c} \text{ in } \dot{a}_i \xrightarrow{A} \dot{b}_0 \xrightarrow{A} \dot{c}$$

is replaced by

$$\dot{a}_i \xrightarrow{A} \dot{c}$$

(see fig. 5). This strategy is repeated until no such chain exists.



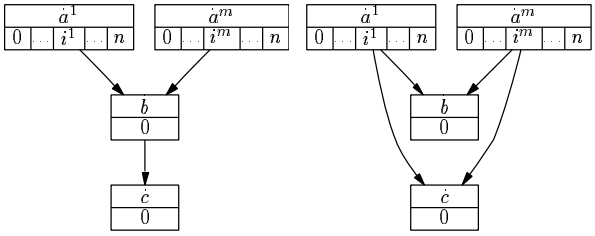
(a) before (b) after

Figure 5. Optimization of sequential allocation-edge chains

2. In this step we treat only merge nodes. The sources of allocation edges starting at merge nodes are moved to the predecessor branch or merge node. So, if $\dot{b}$ is a merge node and $\dot{a}^1, \ldots, \dot{a}^m$ allocate $\dot{b}$ (i. e. $\forall j = 1, \ldots, m \; \exists i \; : \; \dot{a}^j_{ij} \xrightarrow{A} \dot{b}$) and $\dot{b}$ allocates $\dot{c}$ (i. e. $\dot{b}_0 \xrightarrow{A} \dot{c}$), then $\dot{b}_0 \xrightarrow{A} \dot{c}$ is replaced by

$$\dot{a}^j_{ij} \xrightarrow{A} \dot{c} \;\; (j = 1, \ldots, m)$$

(see figure 6).



(a) before (b) after

Figure 6. Optimized allocation at merge nodes

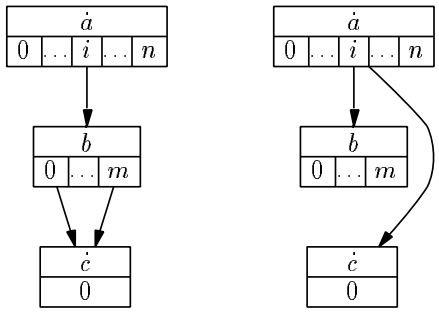
3. In the last step we treat only branch nodes. If a node is allocated by all cases of a branch node, all allocation edges of the node are replaced by one edge starting at the predecessor of the branch node. So, if $\dot{b}$ is a branch node, allocated by $\dot{a}$ (thus $\dot{a}_i \xrightarrow{A} \dot{b}$) and all control flow cases of $\dot{b}$ allocate $\dot{c}$

$$\forall j \in [0, m] : \; \dot{b}_j \xrightarrow{A} \dot{c}$$

then these allocation edges are replaced by one allocation edge

$$\dot{a}_i \xrightarrow{A} \dot{c}$$

(see figure 7).



(a) before (b) after

Figure 7. Optimized allocation at branch nodes

All of the above three steps are repeated in their sequence until no more optimization occurs. The result of this optimization applied to our example graph would produce a

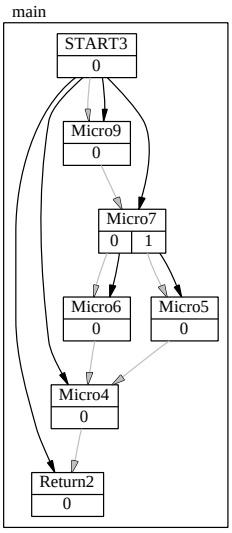


Figure 8. An optimized control/allocation graph

graph as shown in figure 8. We can see here that all nodes but Microframe 5 and Microframe 6 are generated by the start frame of the function. Frames 5 or 6 are generated depending on the `if-then-else` decision. Without consideration of the dataflow which is not yet part of the graph, Micro9, Micro7, Micro4 and Return2 could be executed concurrently.

In the next step we add the dataflow edges to our graph. The dataflow edges are directly derived from the syntax tree generated by SUIF. Together with the dataflow that is obvious in the above C-program there is some additional dataflow caused by some invisible information flow transmissions (e.g. return addresses). Figure 9 shows the appropriate graph. We can also see that routing frames are included into the graph. Routing frames are used as a bypass mechanism for data that is not modified by a thread. In the next optimization step to exploit more parallelism some routing frames that bypass micro frames will be removed. The optimizations locates the following structure in the graph:

$$\dot{a}_i \xrightarrow[x]{D} \hat{b} \xrightarrow[x]{D} \dot{c}$$

If $\dot{c}$ is already allocated when $\dot{a}$ will be executed we can replace both edges by one dataflow edge (fig. 10):

$$\dot{a}_i \xrightarrow[x]{D} \dot{c}$$

After all edges with the above condition have been replaced we will find routing frames without incoming or leaving edges. In this case the routing frame can be removed. Only those routing frames will remain that belong to branch frames, if early allocation was applied as

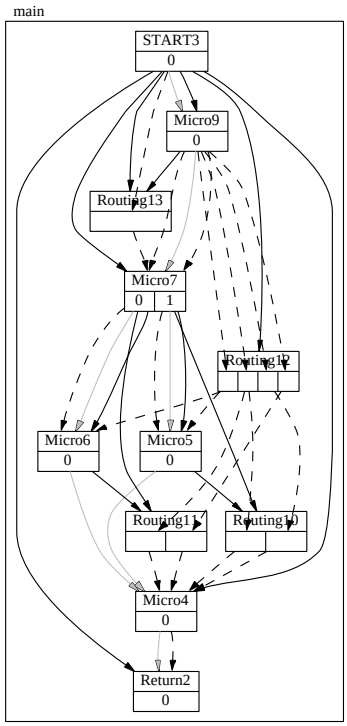


Figure 9. An optimized control/allocation graph with dataflow edges

described above. Figure 11 shows the graph with optimized dataflow.

In the last step we have to care for the following problem: All frames are generated at run-time. Thus, their final memory location is computed at run-time. As the compiler does not know those addresses, it cannot provide the sources of data with the addresses of the sinks. Thus, the addresses also have to be transmitted from the creators of frames to senders who provide other frames with data. As there is no difference for our CDAG between address and data edges,

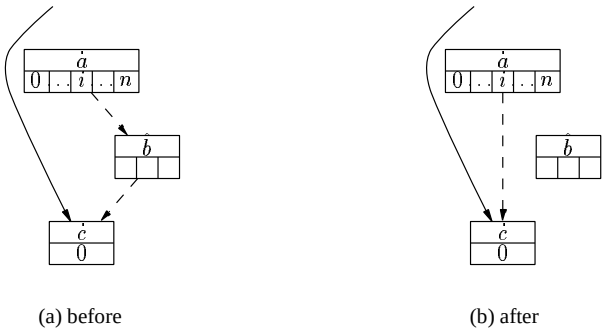


Figure 10. Optimization of dataflow edges

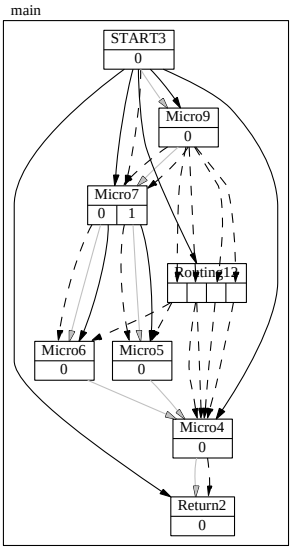


Figure 11. An optimized control/allocation graph with optimized dataflow

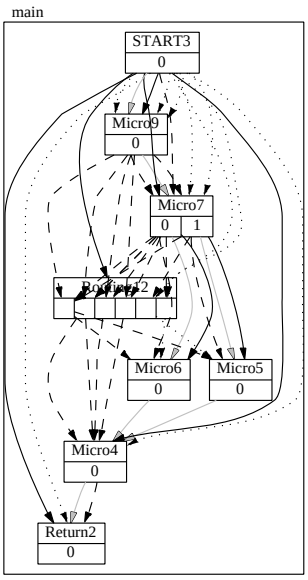


Figure 12. An optimized control/allocation graph with dataflow

we treat them as additional data edges. In figure 12 we can see the complete CDAG.

We can observe in fig. 12 that the micro frames Micro4, Micro5 and Micro6 (and also the corresponding threads) are data independent[2]. After Micro7 has made its decision, Micro4 and Micro5 or Micro4 and Micro6 can be concurrently executed.

2.3 Code generation and execution

Once the optimized CDAG has been created, code is generated for all threads in the graph using the CDAG structure. As the microthreads are pieces of code with strictly sequential control flow, code generation is easy. For the generation of the communicating code sequences and the sequences generating new frames the CDAG is used.

To produce code for different homogeneous machine types or heterogeneous machines we take advantage of the SUIF back end that translates the CDAGs back into C-code. The generated C-code is then compiled to platform specific code by a standard (e.g. GNU) compiler. To run the program, the code of all threads is installed on all machines in the cluster. Additionally the START Frame is generated exclusively on one machine (e.g. the console, where the program has been started). In our example START3 is the start frame. It generates frames Micro9, Micro7, Micro4, Micro2 and Routing12. As there are data dependencies Micro9 and Micro7 will be executed sequentially. After Micro7 has generated Micro5 and Micro6 one of both will be executed concurrently with Micro4. Return2 follows sequentially as it depends on Micro4.

To close this section, we show another, this time containing a loop. The following code sequence is converted by the SDAARC compiler into the CDAG as shown in figure 13.

```
int main()
{
  int i=0,j;
  while (i<10)
  {
    i++;
    j+=5;
    j+=1;
  }
  i=j+1;
}
```

3 Conclusion

In our former papers on the SDAARC architecture we have shown how the automatic data distribution mechanism of COMA architecture can be extended with automatic code distribution. In this paper we show some special compiler techniques that are necessary to generate *migratory* code. The compiler itself is currently in an early state. Correctly and concurrently executable code can be generated but the performance of the code has to be improved. These optimizations will focus on the size of the threads and the allocation time and on array processing.

[2]Micro5 and Micro6 are poor code and could be eliminated. In this example we ignore this fact, because data dependencies between Micro5/Micro6 and Return2 would result in a overly complex illustration.

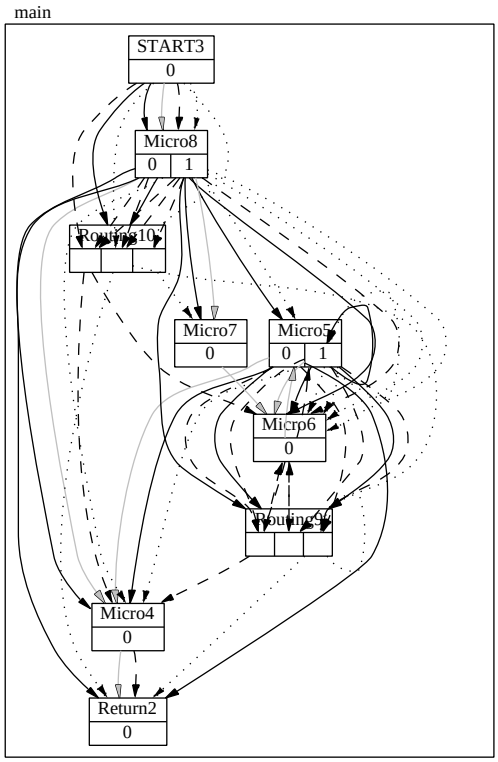


Figure 13. A CDAG with loops

References

[1] M. Annavaram and W. A. Najjar. Comparison of two storage models in data-driven multithreaded architectures. In *Eighth IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 122–129, New Orleans, LA, Oct. 1996. IEEE, IEEE Computer Society Press.

[2] R. D. Blumofe and C. E. Leiserson. Multithreaded computations by work stealing. In *35th Annual Symposium on Foundations of Computer Science (FOCS '94)*, pages 356–368, Santa Fe, NM, Nov. 1994. <ftp://theory.lcs.mit.edu/pub/cilk/focs94.ps.Z>.

[3] E. Hagersten, A. Landin, and S. Haridi. DDM — A Cache-Only Memory Architecture. *IEEE Computer*, 25(9):44–54, 1992.

[4] R. Hipschman. How SETI@home works. <http://setiathome.ssl.berkeley.edu/about_seti/>, 2000.

[5] W. T. S. III, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, and D. Anderson. A new major SETI project based on Project Serendip data and 100,000 personal computers. In C. Cosmovici, S. Bowyer, and D. Werthimer, editors, *Astronomical and Biochemical Origins and the Search for Life in the Universe, Proceedings of the Fifth International Conference on Bioastronomy*. Editrice Compositori, Bologna, Italy, 1997. <http://setiathome.ssl.berkeley.edu/woody_paper.html>.

[6] B. Klauer, R. Moore, and K. Waldschmidt. The Self DistributingAssociative Architecture SDAARC. In *7th Australasian Conference on Parallel and Real-time Systems (PART 2000)*, Sydney, Australia, Nov. 2000. Springer. <http://www.ti.informatik.uni-frankfurt.de/Papers/Adarc/sydney.pdf>.

[7] R. Moore, M. Klang, B. Klauer, and K. Waldschmidt. Combining static partitioning with dynamic distribution of threads. In F. J. Rammig, editor, *Distributed and Parallel Embedded Systems, IFIP WG10.3/WG10.5 International Workshop on Distributed and Parallel Embedded Systems (DIPES '98)*, pages 85–96. Kluwer Academic Publishers, 1999. <http://www.ti.informatik.uni-frankfurt.de/Papers/Adarc/dipes98.pdf>.

[8] R. Moore, B. Klauer, and K. Waldschmidt. Automatic scheduling for Cache Only Memory Architectures. In *Third International Conference on Massively Parallel Computing Systems (MPCS '98)*, Colorado Springs, Colorado, Apr. 1998. <http://www.ti.informatik.uni-frankfurt.de/Papers/Adarc/colosprings.pdf>.

[9] R. Moore, B. Klauer, and K. Waldschmidt. A combined virtual shared memory and network which schedules. *International Journal of Parallel and Distributed Systems and Networks*, 1(2):51–56, 1998. <http://www.ti.informatik.uni-frankfurt.de/Papers/Adarc/barcelona.pdf>.

[10] R. Moore, B. Klauer, and K. Waldschmidt. Tailoring a self-distributing architecture to a cluster computer environment. In *8th Euromicro Workshop on Parallel and Distributed Processing (EURO-PDP 2000)*, Rhodes, Greece, Jan. 2000. IEEE Computer Society Press. <http://www.ti.informatik.uni-frankfurt.de/Papers/Adarc/rhodos.pdf>.

[11] R. Moore, B. Klauer, and K. Waldschmidt. The SDAARC architecture. In *Proceedings of the Ninth Euromicro Workshop on Parallel and Distributed Processing (PDP 2001)*, pages 429–435, Mantova, Italy, Feb. 2001. IEEE Computer Society Press. <http://www.ti.informatik.uni-frankfurt.de/Papers/Adarc/mantova.pdf>.

[12] R. S. Nikhil. A multithreaded implementation of Id using P-RISC graphs. In *Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, pages 390–405, Portland, Oregon, Aug. 1993. Springer Verlag LNCS 768.

[13] P. Stenström, T. Joe, and A. Gupta. Comparative performance evaluation of cache-coherent NUMA and COMA architectures. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 80–91, Gold Coast, Australia, May 1992.