Data Transport Reduction in Move Processors

Marnix Arnold and Henk Corporaal

Delft University of Technology
Department of Electrical Engineering
Section Computer Architecture and Digital Systems
P.O. Box 5031, 2600 GA Delft, The Netherlands
e-mail: marnix, heco@cardit.et.tudelft.nl

Abstract

An important problem in instruction level parallel (ILP) machines is how to handle the many data transports associated with large numbers of concurrent operations. In order to reduce the number of programmed buses needed for those transports, several architectural optimizations are possible, such as feedbacks, chaining and fusing of function units. This paper describes experiments that were performed to find often occurring instruction patterns, in order to identify candidates for these optimizations. Several were found, most notably the three-operand arithmetic unit and the load/store unit with built-in address calculation unit.

Keywords: Pattern detection, operation chaining, special function units, microprocessor architecture

1 Introduction

In the embedded systems applications domain, many algorithms exist that have a high level of instruction level parallelism. One of the main problems that arise when exploiting this parallelism is to transport all the operand and result data of the concurrently executed operations.

Move processors are programmed by specifying data transports rather than operations. An advantage of this is that transport bottlenecks that are not so obvious in conventional architectures are easily identified. The high degree of control over transports also allows many optimizations at compile-time.

The scalability of the Move architecture makes it possible to add transport buses where needed. This extra transport capacity needs to be controlled, however, resulting in a wider instruction word, which in turn leads to more on-chip control buses, wider cache lines and more instruction word pads. These effects cause an increase in chip size (and, hence, chip cost), especially when the design becomes pad-limited.

Rather than just adding raw transport capacity, with all of the abovementioned drawbacks, a more subtle approach may be needed to alleviate the strain on the processor's transport capacity. In this report, three optimization strategies are discussed that make use of the fact that in some cases, data can be transported by other means than programmed buses. In order to identify often performed transports (and therefore interesting points to apply the optimizations), a trace simulation can be performed of a given application, during which the occurrence of specific patterns of operations can be logged.

Some related work is discussed in section 2. In section 3, the three optimization strategies are defined. Section 4 describes the trace simulation experiments that were performed on a set of benchmark applications, in order to determine how and where the optimizations will have the best results. The results of these experiments are presented and evaluated in section 5. The last section features the conclusions and recommendations for further research.

2 Related Work

The results of the research described in this paper can be used as feedback, to get a head-start when searching the design space for the optimal processor configuration for a given application. In that sense, the goals are similar to those of the research described by Onion e.a. in [1]. There is a large difference in approach, however: the experiments by Onion were conducted on a program graph with (unspecified) parallelizing compiler optimizations performed on it, causing the detection of instruction sequences to be dependent on instruction order and register allocation. Our experiments were performed on a true data dependence graph, reconstructed from sequential Move-code. This makes the detection of sequences independent of the compiler's performance.

Phillips e.a. [2] investigated *Interlock Collaps-*

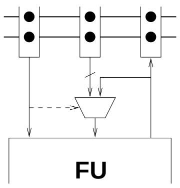
ing ALU's, units that perform two subsequent arithmetic/logic operations in one cycle. Their approach focussed on run-time detection and collapsing of interlocked instructions, and as a consequence the intermediate result should always be calculated (since it may be needed by yet another operation, later during the program). This is done in parallel by a separate unit, something which may not always be necessary (but it is difficult to detect this at run-time). An advantage of our, compile-time based approach is that we can determine beforehand whether we need the intermediate result or not, so that the extra unit may be used for more useful work instead.

3 Transport optimization strategies

In this section, three strategies are described that have a common goal: to reduce the strain on the programmable transport buses, so that their number can be kept as low as possible. These strategies will be named *feedback*, *chaining* and *fusing*.

Feedback

The concept of feedback is a simple one: use a bypass path from the output of a functional unit (FU) to one of its inputs. An opcode bit must be added to specify whether the bypassed value must be used. This optimization relieves the transport bus that would have been used to transport the result value to the input.



The following example illustrates the use of feedback in an addition unit.

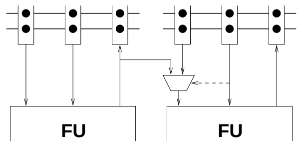
`RISC-code:`
$Add\ \ r_1,\ \ r_2,\ \ r_3$
$Add\ \ r_5,\ \ r_4,\ \ r_3$

`Original Move-code:`
$r_1 \rightarrow\ \ add\_T,\ \ r_2 \rightarrow\ \ add\_O$
$r_4 \rightarrow\ \ add\_T,\ \ add\_R \rightarrow\ \ add\_O,\ \ (add\_R \rightarrow\ \ r_3)$
$add\_R \rightarrow\ \ r_5$

`Optimized Move-code:`
$r_1 \rightarrow\ \ add\_T,\ \ r_2 \rightarrow\ \ add\_O$
$r_4 \rightarrow\ \ add * \_T,\ \ (add\_R \rightarrow\ \ r_3)$
$add\_R\ \ \rightarrow\ \ r_5$

Chaining

Chaining is a lot like feedback, in that an FU output value is bypassed to an FU input. The difference is that the FU where the value comes from is a different one than where the value goes to. The additional opcode bit applies here too, and the advantage of chaining is similar to that of feedback.



The following example illustrates the use of chaining of an addition unit and a comparator.

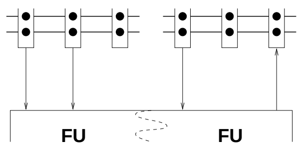
`RISC-code:`
$Add\ \ r_1,\ \ \#25,\ \ r_1$
$Cmp\_eq\ \ r_3,\ \ r_2,\ \ r_1$

`Original Move-code:`
$r_1 \rightarrow\ \ add\_T,\ \ \#25 \rightarrow\ \ add\_O$
$r_2 \rightarrow\ \ eq\_T,\ \ add\_R \rightarrow\ \ eq\_O,\ \ (add\_R \rightarrow\ \ r_1)$
$eq\_R \rightarrow\ \ r_3$

`Optimized Move-code:`
$r_1 \rightarrow\ \ add\_T,\ \ \#25 \rightarrow\ \ add\_O$
$r_2 \rightarrow\ \ eq * \_T,\ \ (add\_R \rightarrow\ \ r_1)$
$eq\_R \rightarrow\ \ r_3$

Fusing

This is the most far-reaching optimization. Two (or more) operations are combined into a single, new operation. This way, no intermediate values will be transported over any transport buses. As a consequence, the intermediate values cannot be accessed by other operations, but in many cases that is not necessary. Well-known examples of fused operations are the multiply-add and the 3-input add.



The following example illustrates the fusing of a multiplier and an addition unit into a multiply-add (mad) unit.

`RISC-code:`
$Mul\ \ r_3,\ \ \ r_1,\ \ r_2$
$Add\ \ r_5,\ \ r_3,\ \ r_4$

`Original Move-code:`

$r_1 \rightarrow \ mul\_T, \ r_2 \rightarrow \ mul\_O$
...
$r_4 \rightarrow \ add\_T, \ mul\_R \rightarrow \ add\_O, \ (mul\_R \rightarrow \ r_3)$
$add\_R \rightarrow \ r_5$

```
Optimized Move-code:
```
$r_1 \rightarrow \ mad\_T, \ r_2 \rightarrow \ mad\_O1, \ r_4 \rightarrow \ mad\_O2$
...
$mad\_R \rightarrow \ r_5$

The applicability of a fused-instruction unit is greatly increased if the unit can also still perform one or more of the operations that the compound-instruction is made up of. For example, a multiply-add unit can also be used to do a multiplication by setting the add-operand to zero, or an addition (albeit a slow one) by setting one of the multiplication operands to one. These special values for some of the operands can be set in hardware, using extra opcodes, or in software, by moving dummy values in as regular operands. The latter case needs extra transport bus capacity, which should be avoided if transport reduction is an issue.

As can be seen from the figures, the three strategies are orthogonal: they can be combined at will.

4 Pattern detection experiments

In order to determine the usefulness of the optimization strategies introduced in section 3, it is necessary to find out how often actual programs can use them. This is done by simulating a set of benchmark DSP algorithms and searching for often occurring instruction patterns. These patterns are candidates for optimization, since the more often a pattern occurs, the more transport capacity can be saved by optimizing it.

The experiments are conducted with a pattern detection and logging program, based on the trace-simulator of the Move compiler [3]. The program works by taking the instructions one by one and finding out which instructions they are dependent on for their operands. This can be repeated recursively to find instruction patterns of any length. The occurrence of each pattern is logged in a tree-shaped data structure. The workings of the program are discussed in more detail below.

Data structures of the pattern detector

There are two main data structures in the pattern detector: one to keep track of the instructions and their dependences (called the operations-array), and one to log the occurrence of the patterns in (called the logging-tree).

The operations array consists of 1024 entries, in each of which information about an operation is stored. Each entry contains:

- a valid field, indicating whether an entry is occupied or not,
- field containing the locations in the array of the operations that supplied the operands for the current instruction (dependencies),
- a not-live field, to indicate whether the value that the operation wrote into its destination register is still live,
- a footprint counter that keeps track of the longest path from the newest node to this one (used during pattern detection).

Coupled to the operations array is the register-writer array. For each general-purpose register that operations can write to (or read from), it contains an index value for the operations array, to indicate which operation last wrote to it.

The logging tree is an n-ary tree structure, where n is the number of different operations that the compiler supports. The structure is allocated dynamically (on demand), since static allocation would require a prohibitive amount of memory for a tree of any significant depth. Each node in the tree contains a counter and a pointer to its subtree.

Processing of the program trace

Each new operation gets an entry in the operations array. Its predecessors (operations it depends on for its input values) are determined by using the registers it reads from as indices for the register-writer array. The values found (indices for the operations array) are stored in the operand-source field of the operations-array entry for the new operation.

If an operand's source is an immediate, then the operand-source field is set to point to zero, a dummy entry in the operations array reserved for immediates. If a value found in the register-writer array is zero, then the register was last written with an immediate value. The operand-source field is also set to point to entry zero in this case: the immediate source value is propagated through the register to the new operation's operand. The same happens if the register-writer value is -1: this means that the register's source is unknown in the context of the current procedure. Unknown sources are also propagated to the operand-source fields. This mechanism is illustrated in figure 2, where the source of r_1 is unknown and the source of r_2 is an immediate. These values are propagated to operation 1 and 2's *op1* and *op2* fields, respectively.

Next, the destination register of the operation is looked up in the register-writer array. If the value found is greater than zero, then this value is used to find the operation that previously wrote this register. This operation's not-live-field is set, since its return value is about to be overwritten by the new operation. The entry number of the new operation is entered into the register-writer array at the appropriate position, and the new operation's not-live-field is set to zero.

Lastly, the valid-field in the operation array entry for the now processed operation is set to 1.

Pruning the operation array

The length of the operation array is limited, and after processing a number of instructions it will be full. A special pruning algorithm is invoked, which operates under the following rule:

An entry in the operation array can be removed if and only if there is no node in the data dependence graph within range S-1 whose write to a register is still live,

where S is the maximum depth of the patterns that should be detected (user defined).

What the rule means is that a node (entry) may not be removed if the possibility exists that a new operation uses its result value or the result value of another node within range S-1. Adding a new operation node would mean that the node under consideration for removal would be part of an instruction pattern with the new node, but this pattern can no longer be detected if the node were to be removed.

In the example of figure 2, node 2 may only be removed if the search depth S is 2 or less. This can be seen easily: if S were greater, then a new instruction that uses r_3 as operand will have a dependence edge to node 3. If patterns of length 3 or longer must be detected, then the new operation will form a pattern with operations 3 and 2. Therefore, entry 2 may not yet be removed. On the other hand, if S were 2 or less, then entry 2 can never be part of a new pattern anymore and may be removed.

The pseudo-code implementation of the operation array pruning algorithm is given below:

```
foreach entry in op. array
  if (entry.not_live)
     entry.not_live = S-1;

for i = 0 to S-2
  foreach entry in op. array
    if (entry.not_live == i)
       set_not_live_bits_of_
           predecessors(i+1)

foreach entry in op. array
  if (entry.not_live == S-1)
     invalidate(entry);
```

The algorithm works as follows: first, all entries whose writes are no longer live are marked with the value S-1. Subsequently, the 'liveness' of the live entries is propagated to their predecessors to a depth of S-2, i.e. to the nodes for which it is possible for new nodes to be attached within range S. All the entries that have not been touched by the 'liveness' of the live nodes are invalidated and can now be overwritten by new entries.

This can be illustrated again using the example of figure 2. The 'liveness' of each live node is propagated to its predecessors to a depth of S-1. If we take S=3, then we can see that the liveness of node 4 will not reach node 1, which is correct because new nodes attached to node 4 will never form a pattern of length 3 which includes node 1. The liveness of node 3, however, does reach node 1, which is correct, since new nodes attached to node 3 do form a pattern of length 3 which includes node 1. Conversely, it can be seen that would we take S=2, then node 1 won't be touched by the liveness of either node 2 or 3, and therefore it may be removed.

Pattern detection and logging

The detection and logging of the instruction patterns is handled entirely recursively. For this report, the detection of instruction sequences (figure 1a) of length 2 and 3, and forking patterns to a depth of 2 (figure 1b) were performed. The results can be found in section 5.

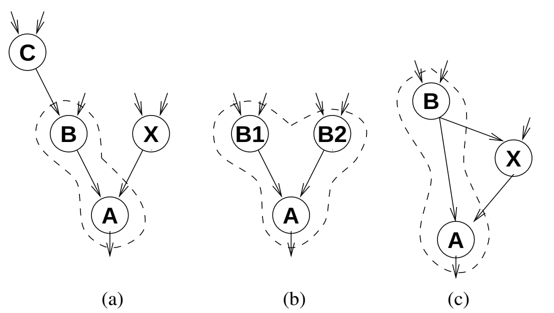


Figure 1: Three classes of patterns.

Patterns such as the one in figure 1c must not be logged, since none of the optimizations discussed in this paper will be of much value here [1]. To ensure this does not happen, a footprint counter is used to keep track of the longest path between two operations. Only patterns that are part of this path (such as AB and BC in figure 1c) must be detected.

For each new operation that is entered into the operation array, the pattern detection procedure is invoked. First, the footprint counters of all entries are set to 0. Then the data dependence graph is traversed recursively to depth S and the footprint counter of each node that is passed is set to the current number of steps taken to reach it, but only if this number is higher than the previous value of the counter. This way, each node that is traversed contains the number of steps to reach it from the new operation node along the longest path.

Then, the tree is traversed a second time, this time to detect the actual patterns. In each node that is passed, the footprint counter is checked. If it is not equal to the number of steps taken until now, then the path isn't followed any further through this node. If the counter value is the same as the number of steps until now, then we know we are on the longest path and we can start logging the instruction patterns we detect.

Each time a node is passed in the operation tree, the counter in the logging tree that corresponds to the operation node's opcode and the path taken through the operation tree until then is incremented.

Figure 2 shows the structure and contents of the logging tree for operation sequences [2] after processing of the code fragment used in the following example.

Example

This example shows how the following sequential Move code fragment is processed by the pattern detection algorithm.

$$
\begin{array}{l}
123 \rightarrow r_2; \\
r_2 \rightarrow add\_O, r_1 \rightarrow add\_T, add\_R \rightarrow r_3 \\
r_2 \rightarrow sub\_O, r_1 \rightarrow sub\_T, sub\_R \rightarrow r_4 \\
r_3 \rightarrow mul\_O, r_4 \rightarrow mul\_T, mul\_R \rightarrow r_3 \\
456 \rightarrow add\_O, r_1 \rightarrow add\_T, add\_R \rightarrow r_5
\end{array}
$$

The data dependence graph of the fragment and the contents of the pattern detection data structures during the processing of this fragment are shown in figure 2.

[1] Bypassing/feedback from B to A would mean that the unit performing A would have to be idle during the execution of X, which is an undesirable restriction. Duplication of node A could be a solution and will be considered in future research

[2] The strategy for forking patterns is similar and won't be discussed here.

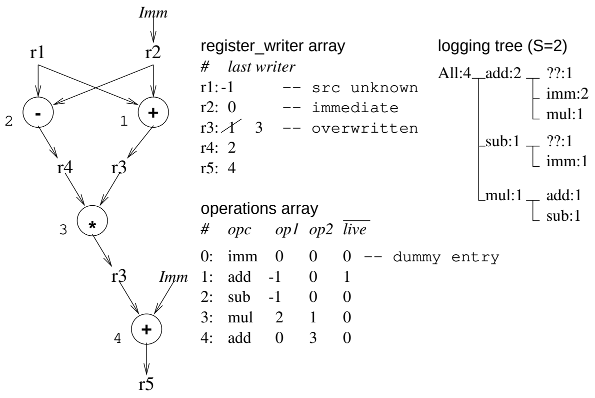


Figure 2: Processing of the example code fragment.

5 Analysis

The pattern detection experiments were performed on a number of audio-DSP, image processing and general-purpose programs. The audio-DSP benchmarks were taken from [4]. The four image processing applications were taken from the Move/Océ case study (MOCCA) and compiled without the special function units described in [5]. In this section, the results of the experiments are presented.

The benchmarks (listed in table 1) were compiled from C source code using the gcc-move frontend compiler (a special version of the GNU c-compiler that generates sequential Move code).

For each of these benchmarks, instruction sequences were detected to a depth of 3. The most frequently occurring ones are listed in table 2. The 'Freq.' percentages are occurrence frequencies for the sub-patterns ('B' and 'BC', as in figure 1) given the root operation ('A'). The percentage in parentheses in column 'A' is the occurrence frequency of that operation in the entire instruction stream. For example (table 2): the operation 'add' makes up 22% of the entire instruction stream. The number of 'add' operations that are preceded by a 'shl' operation is 14.3%. The number of 'add' operations preceded by the 'add-shl' sequence is 4.3%.

Forked patterns (figure 1b) have also been investigated: the results are shown in table 3. It can be seen that the most frequently occurring fork-shaped patterns are really sequences, since one of the sources is an immediate instead of a proper operation.

While some of the numbers certainly look promising (63.5% of all 'add's are preceded by an 'add-add' pattern!), the numbers are much lower when the pattern occurrence frequencies are multiplied by the occurrence frequency of the root operation 'A', to get the frequency of the patterns as part of the entire instruction stream. This is mainly caused by the fact

Domain	Name	Description	Input Data	#ops(stat)	#ops(dyn)
DSP audio	arfreq	Autoregress.freq.estim.	6k audio sample	4330	13612297
	arma			4557	3273217
	equaliz	Equalization	6k audio sample	4329	2990335
	g722	Adapt.differential PCM	6k audio sample	3243	16680630
	instf	Frequency tracking	6k audio sample	4579	3094697
	interp3	Sample rate conversion	6k audio sample	4336	3760517
	intfft2	Interpolate 2:1 with FFT	6k audio sample	10994	2118859
	lms	NLMS algorithm		4455	297537
	mkgwn	Gaussian noise filter		4320	6349646
	mulaw	Speech compression	6k audio sample	1983	320294
	music	Music synthesis		10667	37339605
	osc	Second order oscillator		3300	4413486
	pshift	Kaiser window pitch shift	6k audio sample	12069	2281630
	radproc	Doppler radar processing	800 radar samples	4429	28832634
	rtpse	Spectrum analysis	6k audio sample	4525	2039358
	rfast	Fast convolution with FFT	6k audio sample	4532	3029048
	wavetab	Wavetable music generator		3606	27475240
Image proc. (MOCCA)	minmax	non-linear edge detection	128x107 gray img	23374	2521504
	convol	convolution 3x3 kernel	128x107 gray img	26181	2469719
	floyd	Floyd-Steinberg dithering	128x107 gray img	25451	3542223
	colLUT	interpolated lookup-table	128x107 col. img	24341	8969794
General purpose	compress	file compression	39k text file	19654	2466964
	uncompress	file decompression	19k .Z file	19654	1878311
	a68	68k assembler	stanford.s	61677	2673075
	cpp	C preprocessor	stanford.c	52715	285371
	mpeg_play	MPEG decoding	121k mpeg file	46325	51549969
	djpeg	JPEG decoding	4k jpeg file	56396	3438993

Table 1: The benchmark programs and their characteristics

that no single operation 'A' occurs very often (22% at most, for the 'add' operation).

It therefore is interesting to generate similar tables for *classes of operations* that can be executed by the same hardware. For example: an arithmetic class, containing addition, subtraction and compare operations for both integer and fixed-point values can be mapped onto the same addition hardware. Similarly, the various load and store operations can be performed by the same load/store unit.

As a consequence of this, various sequences of instructions of the same class now also map onto the same fused/chained/bypassed units, thus increasing their utilization potential. Table 4 gives the occurrence frequencies of sequences when the operations are put into the following categories:

- **Arithmetic**: add, addf, sub, subf, negf, eq, eqf, gt, gtu, gtf.
- **Multiply**: mul, mulf.
- **Divide**: div, divu, divf, mod, modu.
- **Load/store**: ld, ldb, ldd, ldh, lds, st, stb, std, sth, sts.
- **Shift**: shl, shr, shru.
- **Logic**: and, ior, xor.
- **Control**: jump, trap, call
- **Other**: all instructions not included in the previous categories.

6 Evaluation

Pattern detection experiments were performed on a number of applications from the digital signal processing and general purpose domains, to see which sequences of operations are most often executed. By grouping similar operations into classes (corresponding to potential function unit types), the number of transports executed between these types of function

A	B	Freq.(%)	C	Freq.(%)
add	add	71.7	add	63.5
(22.0%)			imm	60.9
	shl	14.3	imm	12.7
			add	4.3
lds	add	69.9	imm	47.5
(15.9%)			add	45.9
	imm	26.3		
mulf	lds	69.6	add	42.8
(7.9%)			imm	11.4
	subf	14.3	lds	6.0
	mulf	11.2	divf	4.9
sts	add	60.6	imm	41.3
(6.8%)			add	39.8
	lds	39.6	add	21.7
	addf	27.1	mulf	15.6
			addf	7.7
gt	add	87.1	imm	68.2
(6.1%)			add	61.8
	imm	67.5		

Table 2: Most frequent instruction sequences

A	B1	B2	Freq(%)
add	imm	add	64.4
(22.0%)	imm	imm [a]	10.3
	shl	add	4.0
mulf	lds	lds	50.0
(7.9%)	lds	subf	10.8
sts	lds	add	31.6
(6.8%)	addf	add	15.5
gt	imm	add [b]	28.5
(6.1%)	add	imm	28.3
	add	lds	6.8
addf	mulf	addf	23.1
(6.0%)	mulf	mulf	18.9
	lds	lds	16.8

Table 3: Most frequent instruction forks

[a] This should have been resolved at compile time, but apparently the GCC compiler has not detected this.

[b] This is actually the 'less-than' operation.

units becomes visible. This is a measure for the distribution of inter-function unit traffic density.

Looking at the data in table 4, it becomes clear that the majority of operations (39.9%) are of the arithmetic type, closely followed by load and store operations (32.4%). Both get most of their input data from other arithmetic operations (55.2% and 60.8%, respectively). This suggests that the arithmetic class is a good candidate for feedback, chaining or fusing

A	B	Freq.(%)	C	Freq.(%)
Arith	Arith	55.2 (22.0)	Arith	46.1 (18.4)
(39.9)			Imm	40.2 (16.0)
	Ldst	17.3 (6.9)	Arith	11.4 (4.5)
	Mul	11.0 (4.4)	Ldst	8.5 (3.3)
Ldst	Arith	60.8 (19.7)	Imm	41.5 (13.4)
(32.4)			Arith	36.4 (11.7)
	Ldst	15.6 (5.1)	Arith	10.0 (3.2)
Mul	Ldst	69.3 (5.1)	Arith	47.1 (3.5)
(7.4)			Imm	17.4 (1.3)
	Arith	21.4 (1.6)	Ldst	9.7 (0.1)
			Arith	7.0 (0.1)

Table 4: Instruction sequence frequencies, by category. Absolute frequencies are given in parentheses

(which has actually been applied in [2]).

The high number of load/store operations that are preceded by an arithmetic operation (60.8%) indicates that memory address calculations occur quite often. Dedicated paths from arithmetic units to load/store units would seem a good idea, or even a load/store unit with a built-in address accumulator (arithmetic fused with load/store).

The numbers in table 4 can be read as an upper bound on the percentage of data transports that can be eliminated from the instruction stream when we apply a certain pattern. For instance, using an Arithmetic-Arithmetic (or Arith3) unit would remove up to 5.5% of all data transports (a quarter of 22%, i.e. all intermediate-value moves). This figure is an upper bound because not all Arithmetic operations can be performed immediately after the (Arithmetic) operation that supplies one of its operands, simply because the other operand may not be ready yet. We've seen in table 3, however, that many Arithmetic operations have an immediate as one of their operands, so in reality we might get quite close to the upper bound.

In order to get the actual transport reduction and speed-up figures, it is necessary to cover the data dependence graph of an application with instruction patterns that correspond to the complex function unit operations. For instance, if we look at the example presented in figure 3, it can be seen that the number of data transports is about 40 [3], and the longest path delay is 10 cycles [4]. If we apply the covering given by the dashed lines, we see that the number of transports

[3] Not counting intermediate register stores that are never used again. The actual number of transports in the scheduled version of the code may differ somewhat, due to the effects of bypassing, register allocation and multicasting.

[4] Taking a latency of 2 cycles for multiplications and 1 cycle for all other operations.

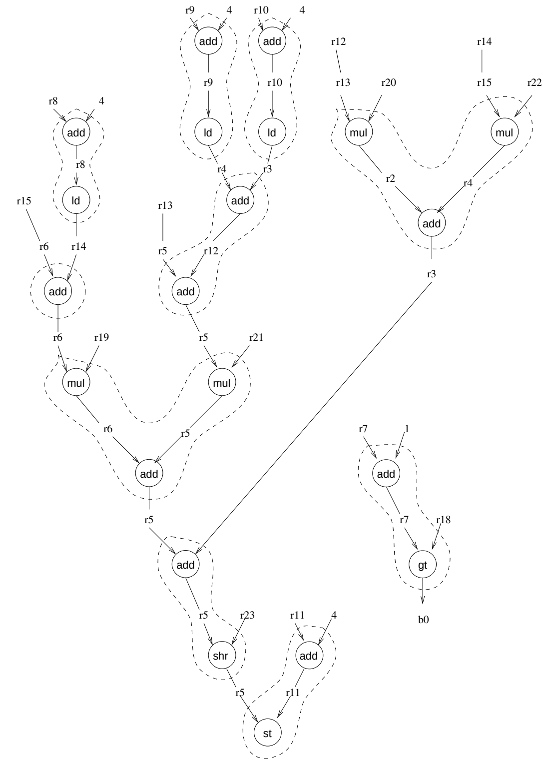


Figure 3: Partial data dependence graph of the convolution algorithm, with complex function unit covering.

is reduced by 11 (about 25%), whereas the longest path delay is now 7 cycles (a speed-up of 30%).

7 Conclusions and Future Research

This paper investigated the applicability of function units supporting more complex operations, built up as combinations of basic, RISC-like operations, in order to reduce the required number of programmable buses and therefore substantially reducing the silicon area of microprocessors. Depending on the implementation of the complex operations, a significant cycle count improvement can also be obtained.

We have demonstrated a tool for the automatic recognition of instruction patterns and their occurrence statistics. Experiments with this tool have revealed many opportunities to apply bypassing, chained and fused function units.

In a future research effort, a data dependence graph (DDG) covering algorithm will be developed that will measure the benefits (both in terms of speed and transport reduction) of these units. By taking the delay of the longest path of the optimal cover (using the new units) of the DDG and comparing it with the original delay (without special function units), the maximum attainable speedup will be calculated for all applications.

References

[1] Frederick Onion, Alexandru Nicolau, and Nikil Dutt. Compiler Feedback in ASIP Design. Technical Report 94-2, University of California Irvine, Irvine, CA 92717-3425, September 1994.

[2] Stamatis Vassiliadis, James Phillips, and Bart Blaner. Interlock Collapsing ALU's. *IEEE Transactions on Computers*, 42:825–839, July 1993.

[3] Jan Hoogerbrugge. *Code Generation for Transport Triggered Architectures*. PhD thesis, Delft University of Technology, February 1996.

[4] P.M. Embree. *C Language Algorithms for Real-Time DSP*. Prentice Hall, 1995.

[5] Marnix Arnold, Reinoud Lamberts, and Henk Corporaal. High Performance Image Processing using TTAs. In *ASCI'96*, pages 150–155, Lommel, Belgium, June 1996.