HAL open science

Register Reverse Rematerialization

Mouad Bahi, Christine Eisenbeis

To cite this version:

Mouad Bahi, Christine Eisenbeis. Register Reverse Rematerialization. [Research Report] 2011. inria-00607323

HAL Id: inria-00607323

https://inria.hal.science/inria-00607323v1

Submitted on 8 Jul 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Register Reverse Rematerialization

Mouad Bahi
INRIA Saclay - Ile-de-France
LRI, Univ. Paris-Sud 11
Orsay, France
mouad.bahi@inria.fr

Christine Eisenbeis
INRIA Saclay - Ile-de-France
LRI, Univ. Paris-Sud 11
Orsay, France
christine.eisenbeis@inria.fr

ABSTRACT

Reversible computing could be in more or less long term mandatory for minimizing heat dissipation inherent to computing. It aims at keeping all information on input and intermediate values available at any step of the computation. Rematerialization in register allocation amounts to recomputing values instead of spilling them in memory when registers run out. In this paper we detail a heuristic algorithm for performing reverse register materialization and we use the high memory demanding LQCD (Lattice Quantum ChromoDynamics) application to demonstrate that important gains of up to 33% on register pressure can be obtained. This in turn enables an increase in Instruction-Level Parallelism and Thread-Level Parallelism. We demonstrate a 16.8% (statically timed) gain over a basic LQCD computation. Basic ideas of the algorithm and experimental results were already presented in a poster of another conference.

Keywords

Register pressure, spill code, rematerialization, reversible computing

1. INTRODUCTION

In this paper we revisit register allocation issues from the reversible computing angle.

While being a very old computer science problem, register allocation is always an important issue in architectures where memory access time and communication time are ever and ever increasing with respect to computing time. This work is done in the framework of a project that aims at designing architecture and program for the LQCD (Lattice Quantum ChromoDynamics) application. The target is a sustained PetaFlops(10^{15} Float operations per second) and we know that one or two orders of magnitude must be gained if we want that this sustained speed is obtained in 2 or 3 years from now.

The lattice of sites on which computations are performed is a 4D lattice that is splitted into sublattices, each of which is managed by one processor. Since LQCD is highly communication demanding it is crucial that as few processors as possible are used, that means that parallelism within processors such as *instruction level or thread level parallelism is maximized*. Since LQCD is also memory demanding it is important that the least number of data are stored in memory or equivalently the largest number of useful data. That means first that intermediate values be kept in lowest levels of memory - registers if possible - and therefore that they have short lifetimes or can be *recomputed from other values*. Second that the vertical memory hierarchy is stressed as little as possible: we have to avoid spill operations that store intermediate value in the memory and reload it on demand. This means that we have to *minimize spill code*. A very important assumption that we make throughout this paper is that we consider that communication more than computation matters. In other words we consider that computation is (almost) for free.

Hence in this program we can see that register allocation is a still delicate and critical issue that must in no ways be left aside as it is the basic bottleneck that conditions the whole performance.

Reversible computing has a lot of applications in computer science [6, 9, 2] but as far as high performance architectures are concerned it has important impact on one of the most important barriers that designers are facing, namely energy dissipation [3] (besides power and failures). Reversible computing has a lot to do with the classical issue of trade-off between data storage and data recomputing. In reversible computing no information is ever lost, every past or future intermediate data value can always be retrieved from any point in the program.

In register allocation one can use rematerialization instead of spilling, meaning that we recompute some value v instead of keeping it alive. Recomputation of v is performed from values still stored in registers, recomputation is done in the same way as specified in the program. But there is a part of information on v carried by other values w that *were computed* from v. Hence this gives new opportunities for recovering the v value: undoing the computation from the w values, or in other words, reversely computing v.

Therefore the question that we address in this paper is *whether rematerialization by reverse computing – reverse materialization – can help improving register allocation*. We develop a heuristic for rematerialization-based register allocation through reverse computing and demonstrate important

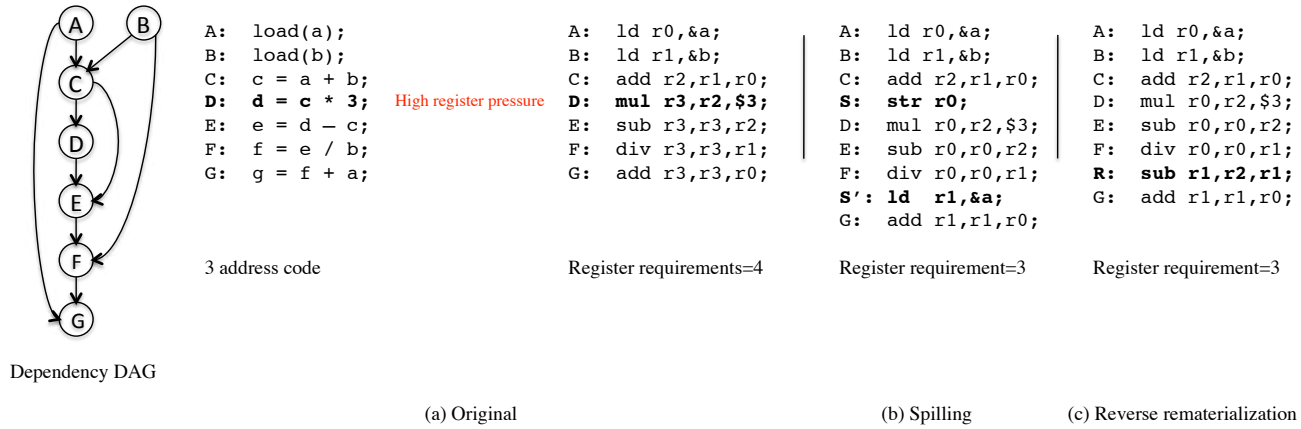


Figure 1: Reducing register pressure using reversible rematerialization.

gains over a kernel of LQCD computation.

2. RECOMPUTING VS STORAGE

In this section, we present our approach about recomputing to minimize load/store operations when we achieve high register pressure. We consider a DAG [1] of operations, typically the Data Dependency Graph (DDG) of instructions within a basic block, see the part (a) of Figure 1. Nodes of the graph are instructions denoted by the name of the variable carrying the result. This makes sense as two different nodes need to be treated as two different variables. We make the important hypothesis that they can be made `reversible`.

2.1 Reversible Operations

In general, a boolean function $f(x_1, x_2, ..., x_n)$ with n input boolean variables and k output boolean variables is called reversible if it is bijective. This means that the number of outputs is equal to the number of inputs and each input pattern maps to a unique output pattern. Based on that we make the abstract approximation that the operations in the DAG are reversible in the following sense: for unary operations, they are bijective so that the operand is uniquely determined by the result. For example, consider the increment function, defined from the set of integers Z to Z, that to each integer x associates the integer $y := x + 1$. The inverse function is $x := y - 1$, easily determined from the result uniquely. For binary operations, this means that only one additional value beside the result is needed for recovering both operands from this result and this additional value. This is typically the case of the basic arithmetic operations, like addition $c := a + b$, where (a, b) can be retrieved from (a, c) or (b, c) by a simple subtraction. Hence the `'+'` operation is considered as having two operands and two results.

This is only an abstraction and we are aware of a number of flaws underlying the concretization of this assumption. A detailed discussion of reversibility of machine instructions can be found in [1]. For instance the multiply $'*'$ operation needs at least one additional resulting bit for determining which of both operands was 0 if the result is 0. We can also use versioning in the program generated by the resulting DAG and run the version reversing multiplications only when input data of multiplications are non zero. There are also data precision issues and round-off problems especially with floating point operations. We discuss that in section 4.5. Therefore in our abstract model, when executing a binary operation we have the choice of memorizing the first or the second operand or both, provided that the reverse operation is possible based on the result and memorized operands.

As an illustration, consider the code segment shown in Figure 1(a) with its corresponding pseudo-assembly code and dependence graph in which each node corresponds to a statement in the code segment. This original pseudo code requires four registers. Figure 1(b) shows the same code that returns the same result as the code in Figure 1(a) but with an additional load/store operation. This code requires three registers. Figure 1(c) shows always the same code that returns the same result but with an additional operation. It shows how a reverse computation could minimize register pressure and avoid load/store operations. Thus, four registers are required in the forward computation, three with an additional reversible operation without any additional load/store operation.

2.2 Recomputing and register reuse

Most problems of spill code minimization are NP-complete [7]. Here we have another degree of freedom. We can recompute a value no matter the number of instructions required to recompute it, and as we make the condition that all operations can be made reversible, a value can be recomputed either from source or result operands.

As in [5] and [17] we base our analysis on the *reuse relation*. An edge (u, v) in the reuse DAG means that the v can be stored in the register previously used by the u value - after it is released. The reuse chains specify the actual reuse as defined by the actual register allocation.

An example of *direct (forward) rematerialization* is shown in Figure 2(a). In the example six registers are required to compute the DAG according to the initial reuse graph drawn in (a). Live ranges of A, B, C, D, E and F overlap. But since B and D are alive during the computation of all outputs of C and E respectively, and since C can be directly

[1] Directed Acyclic Graph

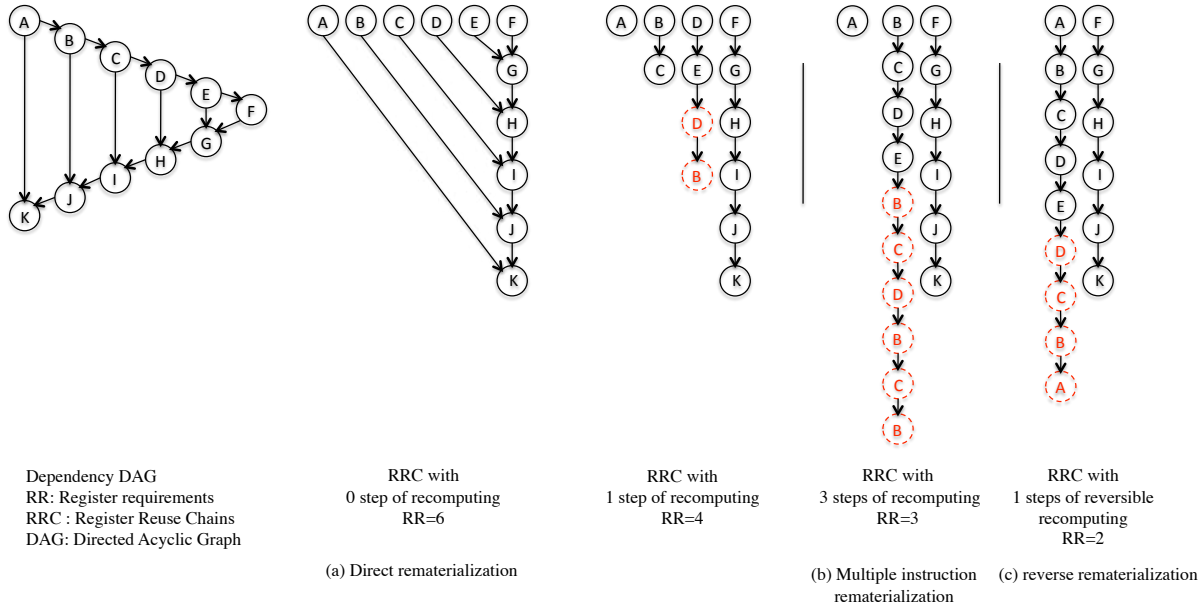


Figure 2: Recomputing vs. Storage.

computed from B, and E from D, we can choose to let C reuse the register of B, and let E reuse the register of D, and recompute later D from C and B from A before computing H and J respectively. This is drawn in the right part of figure 2(a).

We can even more increase register reuse by considering sequences with *more than one instruction* for rematerialization (figure 2(b)). A is alive during all the computation. Thus we can rematerialize B, C, D from A. In this example register requirement is only 3 with 6 additional operations and this could remove 3 spill operations if we had only 3 available registers. As a matter of fact it make sense to use multiple instructions only as far as the execution of the sequence remains negligible with respect to the latency of memory access. In this paper we don't consider this tradeoff and consider computation is free as previously mentioned.

Direct rematerialization aims at avoiding spilling by recomputation. However direct rematerialization is limited because values needed for recomputation have to stay alive and recomputation may take multiple instructions. In contrast considering reverse computing makes rematerialization more attractive because it can both register pressure and number of rematerialization instructions. In Figure 2(c) only 2 registers are required and each recomputed value rematerialized by one instruction which would avoid 4 spills for the whole DAG in the case of 2 available registers. The register pressure is high after computing C, D, E and F causing A, B, C and D to be spilled. A simple way to avoid inserting four load operations before computing H, I, J and K is to rematerialize them from their outputs with one instruction by recomputing D from E, C from D, B from C, and A from B. For minimizing spill code we increase register reuse by reducing values lifetime.

We suggest a hybrid algorithm that both considers direct and reverse computing. The idea is to check at each point when a extra register is needed if a previous value is rematerializable either in a direct or reverse wayin order to consider reusing the register where it is stored.

2.3 Aggressive register reuse

In our work register allocation amounts to determining the reuse chains each of which is allocated in one register [5]. The idea of aggressive register reuse is to enforce register reuse between direct dependent values. Based on the generation of register reuse chains [5] we propose to conside register rematerialization after register allocation. Register requirements is determined from the reuse DAG which indicates which instruction can use a register used by a previous instruction. For the reuse DAG we use the algorithm proposed in [17]. But we use a more aggressive reuse relation by allowing reuse not only for killed variables but also for possibly rematerializable values either directly from the source operands or reversibly from the result operands. This is likely to promote register reuse between dependent values and hence reduce register pressure.

3. REMATERIALIZATION RULES AND GUIDELINES

Rematerialization should be done after register allocation for two reasons:

- Before register allocation there is no information about actual register pressure, register requirement and excessive register demands, which makes rematerialization decision very difficult to make.
- Rematerialization decision before register allocation would create extra dependencies and possibly extend liverange of inputs of the rematerialized value, and this could increase register pressure.

All information regarding excessive registers, rematerializable values, etc. can be extracted from register reuse chains. Berson and all [5] showed that excessive register demands can be better determined using register reuse DAG.

Instruction scheduling and register allocation are very mutually dependent and it is better to manage them in a common framework. This is the aim of the reuse chain framework. Berson and all [5] proposed a register allocation algorithm based on register reuse chains hence solving scheduling and register allocation simultaneously. Zhang [17] showed that register allocation based on a register reuse chains approach requires fewer registers on average than traditional register allocation algorithms based on graph-coloring algorithms.

The register rematerialization can be decomposed into four passes:

- a) Building register reuse chains.
- b) Detecting excessive registers.
- c) Discovering rematerializable values.
- d) Graph transformation.

3.1 Building register reuse chains

The first phase decomposes the input data dependency graph $G = (V, E)$ where V is the set of nodes and E the set of edges in G into reuse chains. Each chain contains values that can share the same register. We start by considering register reuse between dependent values and then between independent values. For each node we identify possible reuse nodes through the relation "*can reuse*". If the live-range of a variable does not overlap the range of another variable they can share the same register. Formally:

$$(u, v) \in V^2,\ u\ can\ reuse\ v \text{ iff } output(v) \cap dependent(u) = \phi$$

where $output(v)$ is the set of direct dependent nodes of v denoted by a direct edge from v in the graph G and $dependent(v)$ is the set of all descendant nodes of v. In other words it is the set all nodes such that there is a path from v to these nodes. In the following we denote also $input(v)$ the set of direct connected nodes by outgoing edge to v and $independent(v)$ the set all nodes such that there is no path between v and these nodes.

3.1.1 Register reuse between dependent values

This is an iterative algorithm. In order to create less additional dependencies we start by first defining register reuse between dependent values. The initial reuse node has to be the earliest ultimate killing (EUK) node. EUK of a node is the earliest node where the lifetime of the values residing in the node is guaranteed to be over [17].

$$k \text{ is } killer \text{ of } v \text{ iff } \forall u \in output(v),\ k\ depends \text{ of } u$$

and

$$q \text{ is } EUK\ node \text{ of } v \text{ if } q \text{ is } killer \text{ of } v\ \wedge\ \forall k\ killer \text{ of } v, k \neq q\ k\ depends \text{ of } q \vee q \text{ is } independent \text{ of } k$$

In other words, if k or q are ready to be computed this means that all theirs inputs which correspond to direct dependent nodes of v, are already computed and as a result v is killed. To build the initial register reuse chains we associate to each node its EUK node if it exists, this allows generating register reuse chains without any additional dependency.
If the EUK of node v does not depend directly to v, we search a closer reuse node, and if two reuse nodes have the same distance from v we compare theirs live-ranges and we choose the latest one. The reduction algorithm visits all nodes that can reuse v, if v has no reuse node we choose the first visited node otherwise we compare the visited node with the actual reuse node. The criteria of comparison are the live-range and dependencies between reuse nodes. .

If the number of register reuse chains is still greater than the number of available registers we proceed in order to reduce the number of chains by defining new reuse nodes from the set of independent nodes.

3.1.2 Register reuse between independent values

Independent values correspond to nodes for which no dependency path between both exists. In contrast of reuse chains correspond to sequences of nodes that can share the same register in some computation. The function *CannotReuse* checks if there is no constraint that prevents reusing a node. If a node is dead and has no reuse node it can be reused by an independent node to reuse it if the latter does not reuse any register and if there is no dependency violation.

During the reduction process some dependencies are added to minimize the register pressure and they have to be considered for scheduling. Once the number of chains reaches the number of available registers we stop reduction. The register requirements is the number of chains. Between dependent nodes the relation *CanReuse* checks if a node can reuse an other node without any dependency violation. Between independent nodes, the relation *CannotReuse* checks if there is constraint preventing register reuse. This iteratively adds dependencies one by one every time we reduce the initial register reuse chains. The register reuse chains are built thanks to the algorithm of register reuse shown in Algorithm 1.

3.2 Detecting excessive registers

Excessive register demands arise when the number of values simultaneously live exceeds the number of available registers or the target number that we are seeking. The register reuse chains identify excessive sets that represent values whose scheduling requires more resources than are available, since scheduling and registers allocation are solved simultaneously, the order in which values are computed is known and as a result excessive demands for registers can be determined. The excessive sets are then used to drive reduction of the excessive demands for registers and rematerialization is used to reduce register demands.

3.3 Discovering rematerializable values

In general, a value stays live because it is used more than once. While it is not used by all dependent operations, it is live and the register of this value cannot be reused. A value v might be rematerializable by a direct operation from source

Algorithm 1: Algorithm of register reuse

```
notation: liverange(p) ≻ liverange(q) if p stays live after
the last use of q
for each node v in the Graph G:
if EUK(v) ∈ output(v) then
    Reuse(v) ← EUK(v)
if (EUK(v) ∉ output(v)) ∨ ({EUK(v)} = ϕ) then
    get P | ∀p ∈ P, p ∈ output(v) ∧ p can reuse v
    if |P| ≥ 1 then
        Reuse(v) ← GetLastLiverange(P)
    else
        get Q | ∀q ∈ Q, q ∈ dependent(v) ∧ q can reuse v
        if |Q| ≥ 1 then
            for all (a, b) ∈ Q^2 do
                if a depends of b then
                    Reuse(v) ← b
                else
                    if b depends of a then
                        Reuse(v) ← a
                    else
                        if liverange(a) ≻ liverange(b) then
                            Reuse(v) ← b
                        else
                            Reuse(v) ← a
        else
            get R | ∀r ∈ R, r ∈ independent(v) ∧
            r can reuse v
            if (|R| ≥ 1) then
                for all (a, b) ∈ R^2 do
                    if a depends of b then
                        Reuse(v) ← b
                    else
                        if b depends of a then
                            Reuse(v) ← a
                        else
                            if liverange(a) ≻ liverange(b)
                            then
                                Reuse(v) ← b
                            else
                                Reuse(v) ← a
```

operands, or by a reverse operation from the result and the rest of operands of the operation.

v is directly rematerializable iff $\forall p \in input(v) \;\Rightarrow\; p$ is *live*

v is reversibly rematerializable iff $\exists q \in output(v)\; \forall p \in input(q)$ s.t. $p \neq v \;\Rightarrow\; p$ and q are *live*

We call *R-input*(v) the set of sets of operands from which v can likely be recomputed either by a direct or reverse operation.

v is rematerializable by multiple instructions iff $\exists S \subset$ *R-input*$(v)\; \forall p \in S \;\Rightarrow\; p$ is *live* or p is *rematerializable*.

3.3.1 Rematerialization decision

In general, an early rematerialization decision (which variable must be used and rematerialized after) before register allocation is definitive and will not be undone later. It might increase register pressure by extending lifetime of inputs of the rematerialized value, which is the length of the longest path from its definition to its last use. The rematerialization decision is efficiently controlled in our approach. It is performed after register allocation and it manages at the same time both rematerializable and excessive nodes of the graph. We first find all rematerializable values and compare theirs live-ranges with the ones of excessive variables. For each rematerializable value we choose the appropriate value that can reuse it and does not prevent its recomputing. Based on register reuse chains we give conditions for a value u to reuse the register of value v in the case of high register pressure and recompute v later when we achieve low register pressure, knowing that live-ranges of u and v overlap.

$\exists x \in ExcessiveRegisters(G),\; v$ is rematerializable $\wedge\; x$ can reuse v iff $\exists S \subset$ *R-input*$(v)\;\; \forall p \in S \Rightarrow\; p$ stays live after x

If at the next use of v we still have high register pressure, we add the condition $\quad output(x) \cap output(v) = \phi$

3.4 Graph Transformation

Once rematerialization decision is made we proceed to the graph transformation of the original DDG. Graph transformation, or graph rewriting, consists in rules for creating a new graph out of an original graph: we insert a copy of the rematerializable value with all edges from its new inputs, and move all edges from the rematerializable value to nodes calculated after the excessive node (new reusing node) to the new copy node.

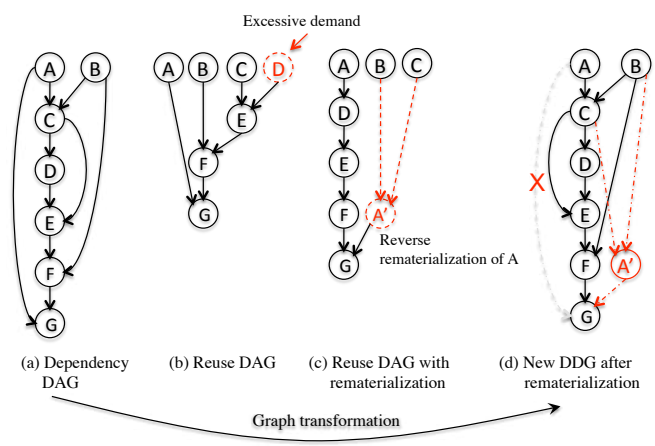


Figure 3: Graph transformation.

The algorithm guarantees that the graph transformation does not create any cycle in the new graph. No cycle exists the new inserted value and the excessive value as all inserted edges from the new node are directed to values computed after the excessive value. Hence there is no path between the inserted node and the excessive node. In Figure 3 we show an example of graph transformation for the 3 address code shown in Figure 1, in this example D is the excessive node, A the rematerializable value and A' is the copy value of A rematerialized from B and C. Because we know that A' is

computed after C and B all output edges from A' are to nodes computed after C and B. Therefore there is no path between output nodes of A' and C or B. As a result there is no path between A' and C or A' and B hence no cycle between A' and C or A' and B.

The rematerialization algorithm we propose is iterative, after each graph transformation we run again the algorithm until there is no more possible rematerializable value.

It is important to note that it may happen that sometimes rematerialization reduces only locally the register pressure and not globally. Even though this does not seem to improve a lot since the register count is fixed, this can in fact be exploited for running more computations (parts of DAG, iterations, threads) in parallel.

4. EXPERIMENTAL RESULTS

In this section, we evaluate the effectiveness of our compiler optimization. In order to understand more about the recomputing opportunities we developed a profiling tool that is able to measure the degree of recomputing in an acyclic code. We perform two separate evaluations: **direct rematerialization** does rematerialize a value with a direct operation from its source operands. **Reverse rematerialization** does rematerialize a value in the case of high register pressure with a reverse operation. In both cases rematerializing a value can be done by multiple instructions. We apply our tool on a set of Data Dependency Graphs extracted from the most critical kernel in the LQCD simulation program [8]. The kernel of the code *(Hopping_Matrix)* contains two separate synchronized loops k and l. One iteration of loops k and l contains 768 and 840 operations respectively. Their corresponding data dependency graph contains 872 and 1016 nodes respectively. Our tool uses a register allocator based on register reuse chains. We used this kernel to show that opportunities exist for both direct and reverse rematerialization to significantly reduce spill costs. We performed many experiments with different values for the number of available registers.

4.1 Register Requirements

Both reverse and direct rematerialization techniques reduce the overall register requirements of *Hopping_Matrix* but with different reduction rates and different costs. Table 2 shows the number of register requirements for each benchmark before and after rematerialization by using reverse and direct computing with one and multiple instructions. The percentage gain is shown in table 4, followed by the cost of reversibility given by the number of additional operations. There is no positive impact of direct rematerialization on *_su3_multiply* and *_complex_times_vector* in *Hopping_Matrix (loop k)* or *_su3_inverse_multiply* and *_complexcjg_times_vector* in *Hopping_Matrix (loop l)*, unlike when using reverse rematerialization with one or multiple instructions that helps by reducing the number of register requirements by 16.2% and 33.3% respectively in *_complex_times_vector*, and 7.1% and 14.2% in *_su3_multiply*.

Even in $Hopping_Matrix$, the direct rematerialization is less successful than reverse rematerialization and with highest cost; 54 operations added which represent 7% of the total number of operations in *Hopping_Matrix (loop k)* for a reduction of register requirements of 18.7%. The same number of register requirements reduction is given by using reverse rematerialization with lowest cost, only 33 operations added from a total of 768 operations that represents 4.3%. With reverse rematerialization with multiple instructions we achieve up to 31.2% of reduction. The direct rematerialization is limited due to the multi using of inputs data, which are not rematerializable in a direct way.

By considering all macros and data structures in loop l of *Hopping_Matrix* as elementary elements, the data dependency graph corresponding to one iteration of the loop l is a tree where each node v (except inputs nodes) depends at least on one node u with one output edge, that means v is the only reuse node of u. As the rematerialization depends on the number of reuse of variables, in a graph if there are few cases of node reuse then there are few opportunities of rematerialization. In the loop l, only reverse rematerialization in functions *_su3_inverse_multiply* and *_complexcjg_times_vector* has a positive impact. Table 2 shows a register pressure reduction of 1 register by using reverse rematerialization with one instruction and 2 registers with multiple instructions.

Finally, note that even in the absence of register pressure reduction, reverse rematerialization can reduce the number of variables that are alive simultaneously at some computing step, thus register pressure is reduced locally and therefore load/store operations could be avoided.

4.2 Spill Costs

We measure spill costs statically. For that we compute the register requirements which is the maximum number of variables that are simultaneously alive. When the number of available registers is less than the number of simultaneously alive variables, the register allocator decides which variables should not be stored in registers and load/store instructions are introduced. The number of spill operations depends of the number of variables that exceed the number of available registers, hence reducing the maximum number of simultaneously live variables N by S with R is the number of available registers and $(N - S) \geq R$ means that spill cost is reduced at least by S. For all rematerialization techniques, the number of available register is assumed to be one. Thus, the rematerialization algorithm extracts recomputing and applies rematerialization to reduce register requirements as small as possible.

Table 3 shows the number of static spills using different techniques of rematerialization. The table compares all techniques and shows how reverse rematerialization is more beneficial. For example, it indicates 51 load/store operations for $Hopping_Matrix$ without rematerialization; compared to 45 load/store operations with direct rematerialization and no spill instruction using reverse rematerialization, for 35 available registers.

4.3 Run-Time Performance

The performance improvement from the reduction in explicit spills cannot be determined exactly, but running the two equivalent codes in Figure 4 - a simulation of the above example in Figure 2 - shows a difference in performance up to 40% for a sequence size equals to $100x2^{10}$. This is because for

the first code, the maximum number of simultaneously alive values is the sequence size which is larger than the number of registers creating more spills to memory. Inversely for the second code where the number of simultaneously live values is constant, two registers, and independent of the sequence size.

```
for(i=0;i<SIZE-1;i++)
    A[i+1]=A[i]+ i;
B=A[SIZE-1];
for(i=SIZE-2;i>=0;i--)
    B=B*A[i];
return B;
```

(a) without rematerialization

```
for(i=0;i<SIZE-1;i++)
    A=A+i;
B=A;
for(i=SIZE-2;i>=0;i--){
    A=A-i;
    B=B*A;
}
return B;
```

(b) with reverse rematerialization

Sequence's size	5120	10240	102400	1024000
%gain (double)	+25%	+37%	+40%	+45.5%
%gain (simple)	-6%	+10%	+26%	+30%

Figure 4: Contribution of reverse rematerialization to execution time

4.3.1 Instruction level parallelism

Having more available registers and increasing register reuse using recomputing, allows more independent instructions to be performed simultaneously; therefore pipelining can overlap the cost of recomputing, and improving the instruction level parallelism would reduce cycle time of a program.

Consider the assembly code shown in Figure 5(a) which corresponds to a part of *_su3_multiply* macro. The added operations using reverse rematerialization do not increase the cycle time since they are overlapped with synchronized operations and replace stall cycles. Operation 2 depends on the results of operations 0 and 1, so it cannot be computed until both of them are completed. Knowing that input data stays alive during the whole computation, with only two available registers, the processor stalls for six cycles before computing the result of operation 2. However operations 0, 1, 3, 5, 7, 9 do not depend on any other operation, so they can be calculated simultaneously if there are enough available registers. As can be seen in Figure 5(c), with reverse rematerialization we can use registers of input data to perform the six operations simultaneously and replace most of stall cycles. We rematerialize input data once intermediate values are used. The performance gain in this example is up to 31.7%. In this example we consider that all operations and their inverses have the same cycle time, 6 cycles.

benchmark	available registers	without remat.	rev. remat.	%gain
_vec_times_vec	14	49	44	10.2%
Hopping_Matrix (loop k)	-	4815	4004	16.8%

Table 1: Contribution of reverse rematerialization to increase instruction level parallelism

Table 1 shows the number of clock cycles - statically timed using spu-timing tool of IBM on Cell BE, and the speedup due to reverse computation for some basic computations of LQCD program, given a fixed number of available registers.

4.4 More Opportunities for reverse rematerialization than for direct rematerialization

Any operation is reversible since the register reuse is limited between direct dependent operands, which means that values of reused registers can be retrieved easily from output operands, contrary to the direct rematerialization where the necessary condition is to keep all input values of the operation live.

Also, the direct rematerialization is limited in the fact that program's inputs are not rematerializable, and there is only one way to recompute intermediate values through direct operations, unlike the reverse rematerialization where inputs like intermediate values are always rematerializable and from different instructions since register reuse is limited between direct dependent values. Taking back the example in Figure 2. B can be recomputed in reverse way from two different operations, from C or I and J. However there is only one way to recompute B from A. This effect has to be more precisely measured experimentally and possibly theoretically analyzed with respect to DAG properties.

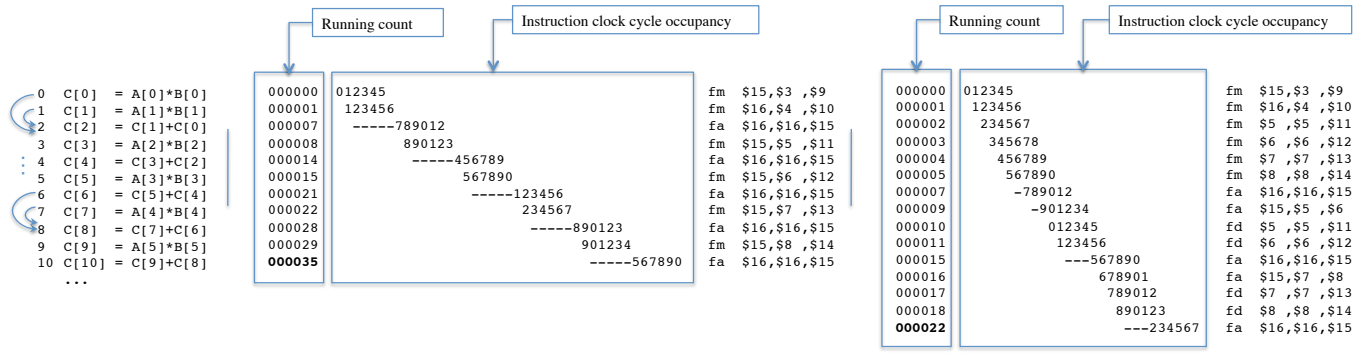
4.5 Inverse Precision

In a static analysis of FORTRAN programs, Knuth [12] reports that 39% of arithmetic operators were additions, 22% subtractions, 27% multiplications, 10% division, and 2% exponentiations.

All numbers expressed in floating point format are rational numbers with a terminating expansion in the relevant base. The number of bits of precision limits the set of rational numbers that can be represented exactly, the error can be related to the decimal place of the right-most significant digit, specifically for multiplication and division where results in general are rounded, so in turn the result of the inverse operation is not exact if the result it self is not, though small errors may accumulate as operations are performed repeatedly. In case of rematerialization with one instruction, at most one operand is inexact, for the multiplication $y*z = x+error(x)$, the error of the inverse operation $\frac{x+error(x)}{y}$ is $\frac{error(x)}{y}$. For the division, $\frac{y}{z} = x + error(x)$, the error of the inverse operation $y = x * z$ is $error(y) = z * error(x)$

But in general, the error propagation given dependent variables each with an error, for addition and subtraction, the precision error in the result is given by: $error(x) = error(y)+ error(z)$ for the operation $x = y + z$. For multiplication and division, the maximum error in the result is given by: $error(x) = error(y) * z + error(z) * y + error(y) * error(z)$. Usually $error(y) << y$ and $error(z) << z$ so that the last therm is much smaller than the other terms and can be neglected. Formally we write more compactly by forming the relative error, that is the ratio of $error(x)/x$, namely $\frac{error(x)}{x} = \frac{error(y)}{y} + \frac{error(z)}{z} + ...$

Even though we did not observe differences between both original and optimized version of one LQCD kernel run, we are aware that this application is very sensitive to data pre-



(a) 3 address code (b) Assembly code: without rematerialization (c) assembly code: using reverse rematerialization

Figure 5: Using reversible computing to increase instruction level parallelism

cision. We have not yet run the whole code that intensively iterates on data and calls this kernel. This will be a meaningful test.

5. RELATED WORKS AND CONCLUSION

Several works have addressed reversibility at the software level. Bishop [6] succeeded to use reversible computing to achieve fail-safety and to protect against both random and systematic faults in underlying compiler and processor hardware. Other works are devoted to the design of reversible programming languages [13], Frank's R [11], but they had to face the issue of trading time complexity with data storage complexity. One elegant method was proposed by Bennett [4] where he models the former problem with a pebble game. Pebbles represent available data at some point of the program. One can add pebbles on some node when there is a way to compute that node with data identified by a pebble in a previous node. One can remove pebbles if there is an alternative way to recompute data required in this node. This is therefore an abstraction of reversible computations that allows analysis of the space and time complexity for various classes of problems, but this simulation operates only on sequential list of nodes. This sequence is broken hierarchically into sequences ending with checkpoints storing complete instantaneous descriptions of the simulated machine. After a later checkpoint is reached and saved, the simulating machine reversibly undoes its intermediate computation, reversibly erasing the intermediate history and reversibly canceling the previously saved checkpoint. Bennett chose the number of pebbles large enough ($n = O(log\ T)$) so that m the number of steps become small.

One of the first works on rematerialization is the work of Briggs et al. in [14], their approach focuses on rematerialization in the context of the Chaitin's allocator [10]. Punjani in [15] has implemented rematerialization in GCC, the experimental results indicated a gain of 1-6% in code size and 1-4% improvement in execution performance. Zhang in [16] proposed an aggressive rematerialization algorithm to reduce security overhead that uses multiple instruction to recompute a value and extends the live-ranges of depending values deliberately to make the values alive through the point of rematerialization.

Most of these previous works target rematerialization between basic blocks and ignore it inside basic blocks, and most rematerialization algorithms are invoked before register allocation which makes rematerialization decision less efficient because of the lack of information concerning register requirements, excessive registers and rematerializable values. This can create extra dependencies to extend live-range of inputs of the rematerialized value and this can increase register pressure.

CONCLUSION

We have presented reverse rematerialization, a novel method for reducing register pressure. Reverse rematerialization takes advantage of the relative cost of computing versus memory access. It recomputes data instead of spilling it. We have found that there may be more opportunities for recomputing a value in reverse path from output operands than recomputing it from its original input operands. In this context reverse rematerialization seems more beneficial. It provides a mechanism to reduce register pressure with a lowest cost than classical rematerialization techniques. Our rematerialization algorithm targets the basic blocks with higher register pressure, it is intended to work after register allocation based on register reuse chains that can provide all necessary information to extract opportunities for recomputation in a graph.

We have seen that reverse rematerialization may improve data parallelism. We are also currently experimenting whether improving register pressure by this method can increase thread-level parallelism typically available in the GP-GPU - general purpose graphical process units. Reverse rematerialization is also an alternative to spilling. Spilling is just storing intermediate values in memory. Conversely we want to see if rematerialization and especially reverse rematerialization could be an alternative to storing unneeded arrays of intermediate values in the memory. This could help exploiting at most as possible the available memory which is one of the bottlenecks of LQCD. The next step would be then to apply it also to communication - recompute rather than communicate.

We also have to extensively check whether precision issues

can be overcome, this will be done on the LQCD application that is a specially well adapted benchmark for that purpose as it requires very high precision at least in some parts.

It is quite interesting to note that when trying to break today's barriers we have to rely on new hypothesis, for instance considering that computation are almost for free. This in turn open new insights on old classical problems such as this register allocation problem, but at the price of opening this precision issue, a new trade-off that we believe is worth of being a topic for future research in high performance computing.

6. REFERENCES

[1] Holger Bock Axelsen, Robert Glückand, and Testuo Yokoyama. Reversible machine code and its abstract processor architecture. In Volker Diekert, Mikhail V. Volkov, and Andrei Voronkov, editors, *Computer Science – Theory and Applications. Proceedings*, volume 4649 of *Lecture Notes in Computer Science*, pages 56–69. Springer-Verlag, 2007.

[2] Henry G. Baker. Nreversal of fortune - the thermodynamics of garbage collection. In *Proceedings of the International Workshop on Memory Management*, pages 507–524. Springer-Verlag, 1992.

[3] C. H. Bennett. Logical reversibility of computation. *IBM J. Res. Dev.*, 17:525–532, November 1973.

[4] C. H. Bennett. *Time-space tradeoffs for reversible computation*. SIAM J. Comput, 1989.

[5] David A. Berson, Rajiv Gupta, and Mary Lou Soffa. Ursa: A unified resource allocator for registers and functional units in vliw architectures. In *PACT '93: Proceedings of the IFIP WG10.3. Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 243–254. North-Holland Publishing Co., 1993.

[6] P. G. Bishop. Using reversible computing to achieve fail-safety. In *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, 1997.

[7] Florent Bouchez. *A Study of Spilling and Coalescing in Register Allocation as Two Separate Phases*. PhD thesis, ENS Lyon, 2009.

[8] A. Shindlerb C. Urbacha, K. Jansenb and U. Wenger. Hmc algorithm with multiple time scale integration and mass preconditioning. *Computer Physics Communications*, 2006.

[9] Christopher D. Carothers, Kaylan S. Perumalla, and Richard M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. In *Proceedings of the thirteenth workshop on Parallel and distributed simulation*, PADS '99, pages 126–135, Washington, DC, USA, 1999. IEEE Computer Society.

[10] G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, New York, NY, USA, 1982. ACM.

[11] M. P. Frank. *The R programming language and compiler*. http://www.cise.ufl.edu/~mpf/rc/memos/M08/M08_rdoc.html, 1997.

[12] D. E. Knuth. An empirical study of fortran programs. In *Software: Practice and Experience, Volume 1, Issue 2*, pages 105–133, 1971.

[13] C. Lutz and H. Derby. *Janus: a time-reversible language*. http://www.cise.ufl.edu/~mpf/rc/janus.html, California Institute of Technology, 1982.

[14] L. Torczon P. Briggs, K. D. Cooper. *Rematerialization*. Conference on Programming Language Design and Implementation, 1992.

[15] Mukta Punjani. Register rematerialization in gcc. In *GCC Developers' Summit 2004*, June 2004.

[16] Tao Zhang, Xiaotong Zhuang, and Santosh Pande. Compiler optimizations to reduce security overhead. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 346–357, Washington, DC, USA, 2006. IEEE Computer Society.

[17] Yukong Zhang, Young-Jun Kwon, and Hyuk Jae Lee. A systematic generation of initial register-reuse chains for dependence minimization. *SIGPLAN Not.*, 36(2):47–54, 2001.

benchmark	without. remat.	direct remat.	reverse remat.	reverse remat. multiple. inst.
_complex_times_vector	6	6	5	4
_complexcjg_times_vector	6	6	5	4
_su3_multiply	14	14	13	12
_su3_inverse_multiply	14	14	13	12
Hopping_Matrix (loop k)	48	39	35	33
Hopping_Matrix (loop l)	48	48	47	45

Table 2: Contribution of reverse rematerialization to minimize register requirements

benchmark	available registers	without remat.	direct remat.	reverse remat.	reverse remat. multiple inst.
_complex_times_vector	5	3	3	0	0
	4	6	6	3	0
_complexcjg_times_vector	5	3	3	0	0
	4	6	6	3	0
_su3_multiply	13	3	3	0	0
	12	6	6	3	0
_su3_inverse_multiply	13	3	3	0	0
	12	6	6	3	0
Hopping_Matrix (loop k)	39	45	0	0	0
	35	51	45	0	0
	33	57	51	9	0
Hopping_Matrix (loop l)	47	4	4	0	0
	45	12	12	8	0

Table 3: Contribution of reverse rematerialization to minimize spill operations

benchmark	register requirements	gain	direct remat.	reverse remat.	reverse. remat. multiple inst.
Hopping_Matrix (loop k) 768 operations 872 nodes	39/48	18.7%	54	33	33
	35/48	27%	-	45	45
	33/48	31.2%	-	-	153
Hopping_Matrix (loop l) 840 op 1016 nodes	47/48	2.1%	-	8	8
	45/48	6.2%	-	-	94
_complex_times_vector 18 op 26 nodes	5/6	16.7%	-	3	3
	4/6	33.3%	-	-	9
_su3_multiply 66 op 90 nodes	13/14	7.1%	-	3	3
	12/14	14.3%	-	-	21

Table 4: Cost of the reversibility: number of additional operations