Journal of Advanced Computer Science & Technology, *3 (1) (2014) 48-60*
©Science Publishing Corporation
www.sciencepubco.com/index.php/JACST
doi: 10.14419/jacst.v3i1.2153
Research Paper

Register-allocated paging for big data calculations

David William Thomas

Towers Watson Limited, Cambridge, United Kingdom
**Corresponding author E-mail: david@david-dylan.co.uk*

Abstract

Software to support the Monte Carlo method generates large vectors of pseudo-random numbers and uses these as operands in complex mathematical expressions. When such software is run on standard PC-based hardware, the volume of data involved often exceeds the physical RAM available. To address this problem, vectors must be paged out to disk and paged back in when required. This paging is often the performance bottleneck limiting the execution speed of the software. Because the mathematical expressions are specified in advance of execution, predictive solutions are possible – for instance, by treating the problem similarly to register allocation. The problem of allocating scalar variables to processor registers is a widely studied aspect of compiler implementation. A register allocation algorithm decides which variable is held in which register, when the value in a register can be overwritten, and when a value is stored in, or later retrieved from, main memory. In this paper, register allocation techniques are used to plan the paging of vectors in Monte Carlo software. Two register allocation algorithms are applied to invented vector programs written in a prototype low-level vector language and the results are compared.

Keywords: *Big Data, Compiler, Monte Carlo, Paging, Performance, Register Allocation.*

1. Introduction

In the Monte Carlo method, a random sample of inputs is fed through a deterministic set of computations. Software is often used, both to generate large pseudo-random samples of inputs and to perform computations on them [1]. Insurance companies develop and use Monte Carlo simulation models to aid decision-making. Fast execution of these models is particularly important during model development, when the model is iteratively modified and tested.

Software sells more readily if it can be run effectively on existing hardware, so it is commercially better if the software can run on the standard hardware found in a personal computer (PC). Users expect software to make best use of the hardware resources available.

One problem that arises when implementing such software is that the volume of data involved in the computations exceeds the physical RAM available on the host. This leads to vectors of values being paged out to disk to make more memory available, and then a particular vector is paged back in when it is required for a later stage of computation. The movement of vectors between memory and disk is often the performance bottleneck which limits the execution speed of these models.

Tian and Benkrid [2] demonstrated the use of field-programmable gate arrays (FPGAs) to speed up a Monte Carlo simulation, but the need for specialized hardware limits the commercial relevance of the findings. In contrast, this project assumes that the software will run on a conventional PC or server. In addition, using FPGAs could be expected to speed up *processing*, but this would have little value if *disk access* is the bottleneck.

Paging is typically treated as a service provided by the operating system. However, the Monte Carlo models considered here perform deterministic computations which are defined in advance of the model run, making it possible to predict when values will be required in memory, when they can be paged to disk, and what the performance impact of paging would be. The unique contribution made in this paper is the use of a predictive rather than reactive approach to the paging of vectors during a long-running computation. In particular, this paper explores the similarities between this problem and register allocation.

Within a compiler, the register allocator assigns each program variable to a register in the CPU. For best performance, a value is held in the same register throughout its lifetime. This is not always possible, so the compiler must decide when a value should be moved from one register to another or saved to memory and reloaded later. Reducing the number of times a variable needs to be copied between a register and memory can reduce the overall run time of the compiled code, and this continues to be an active area of research.
The next three sections describe register allocation. Then the idea of register-allocated paging is introduced, first theoretically (in section 5) and then practically (in sections 6 and 7). The conclusion (section 8) summarises the findings and suggests further research.

2. Register allocation

2.1. Terminology

This sub-section defines some terms that are commonly used in relation to register allocation. The term being defined is shown in italic text.
A *program* comprises a sequence of *instructions*. Most instructions have one or more operands, and many instructions also *define* a result. These operands and results, similar to variables in a high-level programming language, are known as *pseudos* [3]. A pseudo resulting from one instruction will usually be used as an operand in a later instruction. One instruction is sometimes considered to be made up of two *program points* [4]: one before the instruction and one after. At the first program point the registers contain their input values; at the second program point the result register contains the result.
Instructions serve many purposes, but three particular instructions which are important for register allocators are *store* (copy from register to main memory), *fetch* (copy from main memory to register) and *move* (copy from register to register).
In a program without loops, the *live range* [5] of a pseudo is the sequence of instructions beginning with the first instruction which involves that pseudo (either defines it or has it as an operand) and ending with the last such instruction. When loops are present, every possible control path from the pseudo's definition to any of its uses is considered; its live range comprises every instruction on these paths. Two live ranges *interfere* if they have any instruction in common. In essence, a pseudo is live if its value cannot safely be overwritten, and two pseudos interfere if there is any instruction where they are both live.
To assist with various compiler activities, including calculation of live ranges, instructions are often grouped into *basic blocks*. A basic block is a maximal set of instructions which can only be entered via the first instruction (the *leader* [4]) and can only be exited via the last instruction. Each program entry point (usually there is only one) is also a leader.
A *control flow graph* is a directed graph in which each node represents a basic block and each edge represents a possible flow of control from one basic block to another. It is conventional [4] to add special nodes *Entry* and *Exit*, with an edge from Entry to the program entry point and an edge to Exit from each instruction at which the program might terminate.
The purpose of *register allocation* is to map the (unlimited) set of pseudos onto the (finite) set of registers in such a way that every pair of pseudos whose live ranges interfere is allocated to different registers [6]. Put less formally, the role of a register allocator is to arrange the instructions such that the right value is in the right place at the right time. It deals with two classes of location where a value could be: locations in processor registers are relatively scarce and very fast to access; locations in random access memory (RAM) are relatively plentiful but slow to access.
In order to achieve a correct and complete allocation, the allocator may *spill* a pseudo [7] by inserting a store instruction immediately after its definition, inserting a fetch instruction before each use and replacing the original pseudo operand with the one just loaded. The allocator may move a pseudo by both inserting a move instruction and replacing each subsequent use of that pseudo as an operand with the one defined in the move. This has the effect of *splitting* a live range [8]. The reverse operation, which combines two live ranges into one, is called *coalescing*.

2.2. Applications

The primary application of register allocation techniques is in compilers. In its traditional role, a compiler takes a program coded in a higher-level language and translates it into instructions in a low-level language. Such instructions are understood by the processor (or "chip") which executes the program.
Dowd and Severance [9] describe "a hierarchy of memories using processor registers, cache, main memory, and virtual memory stored on media such as disk". Traditional register allocation considers the movement of data between the top two levels in this hierarchy. Research has also considered the application of similar techniques at the next level down. For example, Li, Gao and Xue [10] used so-called "memory colouring" to allocate arrays to on-chip SRAM. Yang et al. [11] used a similar approach for managing the stream register file in a stream processor. The stream register file is also on-chip RAM, but in a stream processor it is shared between multiple arithmetic-logic units (ALUs).

In this paper, register allocation is applied even further down the hierarchy, viewing RAM as the scarce but fast storage, and disk as the plentiful but slow storage. This is described in more detail in section 5.2.
One application of such a system would be for running a Monte Carlo simulation model, where the vector itself represents the value of some quantity being modelled and each element corresponds to a different simulation. Thus the length of the vector corresponds to the number of simulations in the model. Towers Watson's Igloo™ product provides the ability to design and execute financial simulation models. Igloo may shuffle simulation vectors, either to remove correlation between vectors which represent independent random variables, or to introduce a specified correlation between random vectors which were generated independently. Such approaches require a vector-based solution because it is not possible to evaluate one simulation independently of the others.

2.3. Approaches

Modern approaches to register allocation can be divided into two categories. One category models the register allocation problem as a graph colouring problem, based on a simple and elegant formulation originally proposed by Chaitin et al. in 1981. Research in this category is described in section 3.
The second category contains algorithms which do not involve graph colouring. These alternatives were needed because recent changes in compiler technology have placed additional requirements on the register allocator, so that the whole problem can no longer be solved by simply colouring a graph. Section 4 examines this category in more detail.

3. Register allocation by graph colouring

3.1. Colouring an interference graph

Chaitin's crucial insight [7] was that the decision of how to allocate registers to pseudos can be regarded as a graph colouring problem. The question of how to colour the vertices of a graph such that no two neighbours have the same colour has long been studied by mathematicians. For example, Brooks proved a theorem relating the minimum number of colours required and the maximum vertex degree in 1941 [12]. (The *degree* of a vertex is the number of edges it has.)
Chaitin's approach was to create an interference graph with a vertex for each pseudo and an edge between two vertices when the two corresponding pseudos interfere. If this graph is coloured using k colours, where k is the number of registers, then a valid register allocation has been found – each colour corresponds to a different register.
There are two further problems to address. Firstly, the number of colours required may exceed the number of registers available. Secondly, colouring an arbitrary graph using the minimum number of colours is NP-complete. This second problem is not as serious as it first seems – we don't need to find a minimum colouring, just a k-colouring. Chaitin found a heuristic which will often give a k-colouring if one exists. If the heuristic doesn't find a k-colouring, the program is modified to reduce interference, a new interference graph is constructed for the modified program and the heuristic is applied again. Conveniently, the same program modifications can be applied regardless of whether the k-colouring problem was impossible or just beyond the capability of the colouring heuristic.
Chaitin's colouring heuristic removes vertices from the graph if their degree is less than k, because there will always be a colour available for them. Removing a vertex reduces the degree of its neighbours. If, as a result, the neighbour's degree becomes less than k, this too can be removed, and so on. This process may eventually remove all of the vertices, in which case the original graph can be coloured by working through the vertices, most recently removed first, assigning the first available colour.
Alternatively, the process may result in a graph in which all the vertices have degree k or greater. In this case, a vertex is selected for spilling (based on a cost metric), the program is modified accordingly, and the whole process is repeated for this new program.
Briggs [13] observed that it is not always necessary to spill a vertex whose degree is greater than or equal to k: if some of its neighbours share the same colour, one of the k colours may still be available. He modified Chaitin's algorithm to place the vertex in a list of spill candidates. Spilling is deferred until the colour allocation stage, and happens only when a spill candidate does indeed have neighbours using all k different colours.
Cooper, Harvey and Torczon [14] reported that the time spent building and manipulating interference graphs forms the largest part of the register allocation process, and experimented with two different data structures: one structure used a bit matrix and edge lists, and the other used a hash table. In the latter case, they also evaluated different hash functions. Their conclusion was that different data structures performed better for different programs.
Galinier and Hao [15] proposed a hybrid evolutionary algorithm for graph colouring. Hybrid evolutionary algorithms combine a genetic algorithm (where selected members of a population are "crossed" with one another, and others are mutated, to produce a new generation, and this process is repeated over many generations) with a local search (which tries to move from a good solution to a better one nearby in the search space). Their experimental results showed that their algorithm compared favourably to the best alternatives at the time. Glass and Prügel-Bennett [16] criticised the experiment in [15] for introducing two novel elements – a crossover operator and a local search technique – at the same time. They performed a similar experiment using the crossover operator combined with an established local search technique, and showed that the quality of Galinier and Hao's solution was mainly attributable to the crossover operator.

Mahajan and Ali [17] produced a hybrid evolutionary algorithm specialised to register allocation. The basic results they presented seem to indicate that their algorithm compares favourably with those of Chaitin and Briggs. Earlier, Cooper, Shielke and Subramanian [18] had investigated the use of a genetic algorithm to reduce code size (which is an important consideration for programs on embedded devices), and mentioned the possible use of the same technique to reduce spill cost.

3.2. Splitting live ranges to reduce interference

Live range splitting was another technique investigated by Briggs [13]. Breaking up a long live range into shorter ones reduces interference and can therefore reduce spilling. One disadvantage is that this adds a move instruction at each point where the live range is split, increasing the code size and potentially the run time. Another disadvantage is that it increases the number of vertices in the interference graph.
Briggs advocated (and George and Appel [19] concurred) that live ranges should be split at each basic block boundary.
Fig. 1 illustrates splitting in two simple examples. In the first example, rather than the same variable *x* being used in two connected basic blocks *B1* and *B2*, x_1 is used in *B1* and x_2 is used in *B2*, and a move instruction is added to copy the value from x_2 into x_1 when control flows from *B2* into *B1*.

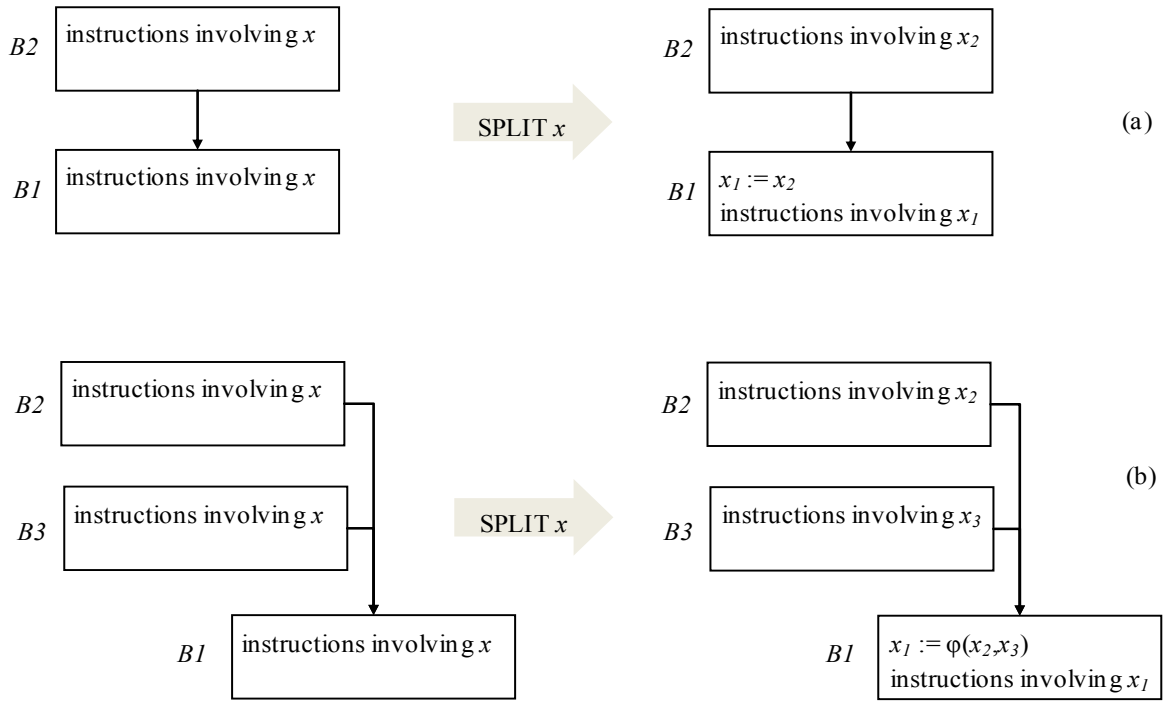

Fig. 1: Two Examples of Splitting a Live Range at Basic Block Boundaries. In (a) There Is One Incoming Control Flow Edge to Basic Block *B1*; in (b) There are Two Incoming Edges.

The situation becomes more complicated in the second example (b), where there is another path into *B1* from *B3*. Before live range splitting, the variable *x* appeared in all three basic blocks and would be allocated a single register. Once it has been split, the value of x_1 may come from x_2 or from x_3, depending on the control flow. Furthermore, x_1, x_2 and x_3 may be allocated to three different registers. The relationship between these variables (and others passed along the same control flow edges) is described using an φ-function, discussed in more detail in section 3.5.

3.3. Coalescing

Optimising compilers use coalescing, independently of register allocation, to eliminate redundant move instructions. This is relevant to register allocation because it replaces two shorter live ranges with a longer one. Indeed, an argument used in favour of live range splitting to ease register pressure *before* register allocation, is that many of the moves inserted could be removed afterwards by an effective coalescer.
Whereas Chaitin [7] and Briggs [13] regarded coalescing as a separate activity, Iterated Register Coalescing [19] interleaves coalescing with other parts of the register allocation process. In essence, it combines the same building blocks used in Briggs's algorithm in a different order to eliminate more move instructions without triggering additional spills.

3.4. Interactions between spilling, splitting and coalescing

At first glance, combining live ranges would seem to increase interference, and therefore be detrimental to colourability. Park and Moon [20] challenged this idea, observing that coalescing two pseudos will reduce the degree of any vertices which neighbour both pseudos in the interference graph, so colourability of the graph could turn out to be better. Their approach is to coalesce as much as possible, and then undo the coalescing of selected vertices, on demand, during the colouring process. The main drawback is the impact this has on the run time of the register allocator: they use experimental results to claim that this may not be prohibitive in practice, but a theoretical complexity of $O(2^n)$ would still deter many practitioners from adopting their algorithm.

Subsequent research [21] [22] has proposed techniques which influence the colouring of the graph in such a way that coalescing, which takes place after register allocation, has more opportunities to eliminate move instructions. Colouring-based coalescing does this by preserving some of the information that would otherwise have been lost during live range splitting [21].

Recall that splitting live ranges after every instruction results in very large interference graphs. Blazy and Robillard [8] recognised that this makes coalescing more difficult. Their technique of live range unsplitting colours the graph and then reverses some of the splitting. In this way, they were able to find optimal coalescing solutions for graphs where no optimal solution had previously been found.

Koes and Goldstein [22] provided a useful review of the important aspects of register allocation – spilling, live range splitting and coalescing – and conducted experiments to assess the contribution of each aspect to the overall quality of an allocation. Their findings, though unsurprising, confirmed that splitting at basic block boundaries is sufficient, that the aspects can be treated separately, and that a simple approach to coalescing is adequate unless code size is important.

3.5. Eliminating φ-functions

As mentioned earlier, the compiler inserts a φ-function to populate a pseudo from one of several other pseudos, with the actual source being determined by the control flow. In fact, a single φ-function may perform this operation on multiple registers at once. The instruction set of a processor does not include such a function, so the compiler must eventually replace these functions with real instructions (moves, or swaps if supported) to accomplish the same result.

Processing φ-functions becomes more complex when a pseudo may flow from more than one basic block into more than one successor (that is, when the control flow graph has a “critical edge”). One way to remove this complexity is to perform edge splitting (not to be confused with live range splitting), where a dummy basic block is inserted on the critical edge [4]. An alternative approach involves moving the implementation of the φ-function along the critical edge [23]. The latter technique also aims to improve the opportunities for both coalescing and efficient scheduling.

3.6. Handling architectural complications

Register allocation techniques based on [7] and [13] assume a simple processor architecture. In practice, processors have additional features that the register allocator must take into account. Here I describe two such features.

Certain instructions require that one (or more) of their operands be in a particular register. This means that the pseudo must either be assigned to that register by the allocator, or it must modify the program to move the value into that register at the appropriate time. In graph colouring, this pseudo is represented by a pre-coloured vertex.

In some architectures, a pair of 32-bit registers might also be used as a 64-bit register under a different name. The registers are said to be *aliased*. The allocator must ensure that each register is only used for one of the two purposes at a time. Smith, Ramsey and Holloway [24] generalised Iterated Register Coalescing to handle aliased registers, and also to handle register classes (for example, one class of registers might be used to hold integers and another class to hold floating-point numbers). Ahn and Paek [25] integrated this technique with optimistic coalescing [20].

4. Alternatives to graph colouring

4.1. Integer linear programming (ILP)

Various researchers, including Appel and George [26], have formulated register allocation as an ILP problem. Initially, this was developed purely for the purpose of research – finding an optimal allocation (however slowly) enabled the researcher to assess how close some other, more practical, algorithm came to optimality ([27] for example). However, Barik et al. [28] identified a number of additional constraints that can be applied to the formulation without destroying the optimality of the solution. These constraints reduce the search space, helping the solver to find a solution more quickly. Furthermore, they showed that the solver often spends much of its time proving the optimality of its solution; it could be stopped earlier if this proof is not required.

4.2. Linear scan

Whilst some researchers have sought higher quality allocations, others have focused on allocation speed. This trend has been driven by the use of just-in-time (JIT) compilation in the Java Virtual Machine and Microsoft .NET Framework architectures, where the cost of compilation is paid by the user at runtime, rather than the developer at build time. Although the allocation produced by a fast algorithm is generally expected to be poorer than by a slower algorithm, register allocators in JIT compilers can use dynamic runtime information, notably the frequency with which a particular basic block is executed, to improve spill decisions. Poletto and Sarkar [30] proposed a linear scan algorithm which "allocates registers to variables in a single linear-time scan of the variables' live ranges." They were able to achieve significantly faster compilation with only a small impact on the run time of the compiled code. Mössenböck and Pfeiffer [30] improved this algorithm to take account of "lifetime holes" – periods when a variable is live but not in active use. Wimmer and Franz [31] found that the linear scan algorithm can be simplified if the program complies with a certain constraint, leading to improved compile speed with no increase in run time. Sarkar and Barik [32] developed Extended Linear Scan, providing convincing empirical evidence that this algorithm is more scalable than graph colouring, and competitive in terms of quality. [32] also gives a cogent critique of the graph colouring approach, picking up on Briggs's observation [13] that graph colouring only tackles a reduced form of the register allocation problem.

4.3. Puzzle solving

To address architectural complications, the prolific researchers Pereira and Palsberg have developed a novel puzzle-solving algorithm [33] which they later enhanced with a technique called Punctual Coalescing [34].

5. Register-allocated paging in theory

5.1. Introducing register-allocated paging

Having described the traditional form of register allocation, I can now consider the central premise of this research: that register allocation solves a problem analogous to the problem of allocating memory for large vectors during the evaluation of complex mathematical expressions, such as those in a Monte Carlo simulation model. In this section I examine some theoretical similarities and differences between the two problems, before moving on to more practical considerations in sections 6 and 7. I continue to use the term "register allocation" for the allocation of registers in a processor, and introduce the term *register-allocated paging* for the large vectors problem.

5.2. Regarding memory as a register file for large vectors

Register-allocated paging makes the assumption that, for any given program (in source code form), the vector operands are all of the same size and the expressions are compiled prior to execution. Before the compiled program is run, an area of RAM is reserved for the program to use, and partitioned such that each section can hold one vector: these are the registers. When a vector must be removed from a register, but is needed later in the program, it is written to a paging file on disk. (If the vector is not needed later, it is left in the register and overwritten when the register is next used as the target of an instruction.).

5.3. Spilling vector pseudos

Spilling a pseudo – that is, storing its value in memory and fetching it again later – has a direct analogy: the paging file replaces memory, and a portion of memory replaces the processor register.

The behaviour of the two kinds of storage differs, however, in the factors that influence the time taken to perform fetches and stores. The time taken to fetch a value from memory depends only on whether that value is already in the processor cache or not. In the case of disk, the disk controller may also have a cache, but when the data is not in the cache, mechanical parts of the disk drive must be moved to the correct location for reading the data. This additional factor is referred to as the *seek time*.

5.4. Program structure and problem size

Many of the concepts related to program structure – basic blocks, live ranges and live range splitting – are identical in both register allocation and register-allocated paging.

My prototype for register-allocated paging has no provision for procedure calls. Although most register allocation techniques are *global* [32], this just means that they work across a whole procedure (in contrast to *local*, which works on a basic block, and *interprocedural*, which works across procedure calls). Thus my prototype's inputs to a global

register allocation algorithm are likely to be considerably larger than the algorithm would encounter in its usual context; larger in both the number of instructions and in the number of pseudos.
Furthermore, my programs may run on a variety of machines. The user specifies the size of the vectors. Hence neither the vector size nor the number of registers (a function of the RAM available) is known when the program is written, so the compilation will have to be done just-in-time – the allocator must be suitable for a JIT compiler. On the other hand, these vector programs can take many hours to run, so the running time of any reasonable algorithm may go unnoticed.

5.5. Coalescing moves of vector pseudos

Much recent research has focused on improved coalescing techniques, which result in programs containing fewer move instructions. Even in traditional compilers, avoiding a move makes only a marginal improvement to the run time of the program. For this research I decided to focus on spilling and to consider coalescing only in terms of its impact on spilling.

6. Prototype of register-allocated paging

6.1. Overview of prototype

To assess feasibility and enable experimentation, I developed a prototype low-level language for processing vectors, two register allocation algorithms and a set of sample vector programs. All development was in C# using the Microsoft .NET Framework.
The low-level language was implemented as a set of classes representing instructions. The instructions target a notional vector virtual machine, which would be implemented in software. Certain types of instruction (notably fetch and store) are relevant to a register allocation algorithm: it needs to add these to implement a spill. For most kinds of instruction, the only information relevant to the register allocation algorithm is which pseudos are involved as operands and which (if any) pseudo is defined. Thus a rich and useful instruction set was not needed for this prototype.
I selected Iterated Register Coalescing (an approach based on graph colouring) and Extended Linear Scan for detailed study and implementation. Iterated Register Coalescing has been used by various other researchers as their basis for comparison, and although there are several refinements to it in the literature, these either focus on improved coalescing behaviour or on architectural features. Extended Linear Scan is designed for just-in-time compilers. Its authors present strong empirical evidence of its superior scalability compared to graph colouring. Both of these purported benefits are relevant to register-allocated paging. Furthermore it is relatively modern (in the context of register allocation) and likely to be close to state-of-the-art. Both algorithms are available in pseudo-code [32] [35].
Initially I wrote a small sample program by hand to accomplish a simple data transformation task. For experimentation I used larger generated programs, all of a similar structure but having different numbers of vector pseudos. These programs were more realistic in scale but less realistic in function. I used these programs to investigate what happens when the selected register allocation algorithms are applied to large-scale programs.
Applying a register allocation algorithm to a program requires certain supporting information about the program and my prototype included some data structures for this purpose. In particular both algorithms required liveness information derived from a control flow graph.
The remainder of this section identifies key decisions I made and the issues I encountered while implementing the prototype.

6.2. Prototype language and virtual machine

The design of the low-level language allowed for both scalar and vector variables. Only the vector variables were subject to register allocation. Scalar variables might be used for control flow (as a loop counter, for example). The language did not include procedures but this does not seriously compromise the research: one of my sample programs could represent a procedural program in which the procedures have been inlined by the compiler, or one of several procedures within a larger program. It does, however, rely on the assumption that the entire program is compiled in one go (in other words, that a programmer does not call a procedure provided to him in compiled form).
Recall from section 2.1 that one instruction can be viewed as two program points. At the first program point the registers contain their input values, at the second program point the result register contains the result, and the transition from one program point to the other is considered to be atomic. The important consequence for a register allocator is that the result of an expression can legitimately be allocated to the same register as one of the operands, if that operand is not required again at a later program point. The same assumption cannot be made for vector registers. In section 2.2 I mentioned that vectors may be shuffled, so the instruction set might include, for example, an instruction in which the result contains the same values as the first operand, but reordered to be correlated with the second operand. In this example the first element of the result vector is not derived from the first elements of the operand vectors. Writing result values into the same register as either of the operands would be incorrect in this case.

The low-level language targeted a notional virtual machine. The experiments described in section 7 were possible without an implementation of the virtual machine, but it is worth considering what such an implementation would look like. One particular concern involves fetch and store instructions. For vectors, these are not directly analogous to the same instructions in a processor: the mechanics of passing a vector from RAM to disk are not the same as the mechanics of passing a single datum from a register to RAM. Specifically, the time taken to write to disk includes time to move the disk heads and rotate the disks (or seek) to the required location. A similar process is required to fetch a vector from disk.
Furthermore the virtual machine will need to keep an index recording the location of each vector pseudo stored, in order to fetch it again later. Where is this index to be kept? Keeping it in memory reduces the amount of memory available to use for vector registers; keeping it on disk increases the time taken to store or fetch because two seeks are required – one to the index and one to the vector itself.

6.3. Algorithm implementations

I coded my Iterated Register Coalescing (IRC) and Extended Linear Scan algorithms to resemble the pseudo-code as closely as possible, and to show clearly where I had interpolated or added to the originals. Sarkar and Barik [32] actually present two algorithms, known as ELS_0 and ELS_1. ELS_0 is only suitable when spilling is unnecessary; my experiments use ELS_1.
Table 1 summarises the main differences in the design of the two algorithms and the impact of those differences on the research.

Table 1: Summary of Algorithm Design Differences

Area of difference	Iterated Register Coalescing	Extended Linear Scan
Space efficiency	Required a bit matrix for edges.	No issues.
Definition of register allocation	Register allocation chooses a register for each pseudo. Live range splitting must be implemented by modifying the program to replace one pseudo with two (or more).	Register allocation chooses a register for each interval. An interval joins two program points. A pseudo's live range can be presented to the algorithm as multiple intervals, without modifying the program. This research used one interval per pseudo for simplicity.
Avoiding dying operand and target being allocated to same register	Add interference edges between the target and the operands of an instruction.	Extend live interval of operands to include the program point after the instruction and extending the live interval of the target to include the program point before the instruction.
Action on spilled pseudos	Rewrite the program, inserting store and fetch instructions to implement spilling decisions. Consequently the algorithm may make more than one pass over the program. Furthermore, later passes may select for spilling a pseudo that was created by a spill in an earlier pass (Ahn and Paek [25] call this "infinite spilling").	Set a flag on the interval to say that it has been spilled. No guidance is given on how to implement the spill in the compiled program.

The most serious limitation I found with Extended Linear Scan concerns its output. The output of ELS_1 comprises, for each symbolic register (Sarkar and Barik's term for pseudo), either the register to which it has been allocated, or a flag to say that it has been spilled. No details are given of how this output should be applied to the program. In my virtual machine, the assumption was made that even a spilled pseudo occupies a register while it is used in an instruction! The algorithm provides no guidance on how to determine which register to use, and I was not convinced that the algorithm guarantees that a register will be available. I assume that the algorithm is designed for instruction sets which allow operands to be read directly from, and results written directly to, a memory location. For my experiments, I worked around this by assuming a fixed limit on the number of operands per instruction and reserving that number of registers for spilled pseudos.

7. Experimental comparison of algorithms

7.1. Overview of experiments

The prototype was used to perform several experiments. Each experiment was designed to assess one practical aspect of applying the algorithms to register-allocated paging.
The key performance indicator for a register allocation algorithm is the speed of the resulting program. Since the prototype does not (yet) produce runnable programs, the number of store and load instructions inserted by each algorithm was compared instead. Where the number of added instructions is reported, the results were identical every time the run was repeated: the same algorithm made equivalent (possibly the same) spill decisions every time.

In the context of a just-in-time compiler, the resources (processing time and space in memory) used by the algorithm are also important. Of these, memory is less of a concern because it is cheaper and easier to scale (within reason). Running times for the algorithms were measured during the experiments. Because I implemented both algorithms for clarity and correctness, not for performance, I have been selective in the timing results I present and I draw only limited conclusions from them.

Where timings are reported, these are the minimum of 5 observations. The reason for reporting the minimum is that the elapsed time is serving as an estimator for the amount of work done by the algorithm, which, for the same input, is the same each time. It is not possible for any actual observation to be smaller than the theoretical smallest run time on that machine, so taking the minimum provides a better estimate of that theoretical quantity than either the average or the maximum. In any case the measurements were stable, so choosing to use the maximum (or average) would not have affected the analysis.

Each experiment comprised a series of runs. A single run involved one input program, one register allocation algorithm and a number of registers. Each run was performed on a PC with an Intel® Core™ 2 Duo 2.99GHz CPU and 3.25GB RAM running Windows XP Professional Service Pack 3, with Microsoft .NET Framework v4.0.

Table 2 summarises the experiments conducted. The remainder of this section presents the results and highlights any points of interest.

Table 2: Summary of Experimental Variables

Experiment	Independent variables	Dependent variables
Effect of program size on spilling	Algorithm selected Number of pseudos in input program	Number of fetch instructions added Number of store instructions added
Comparing algorithms' spill behaviour at different register pressures	Algorithm selected Number of registers available	Number of fetch instructions added Number of store instructions added
Algorithm resource consumption	Algorithm selected Register deficit	Elapsed time to allocate registers Memory usage during allocation

7.2. Effect of program size on spilling

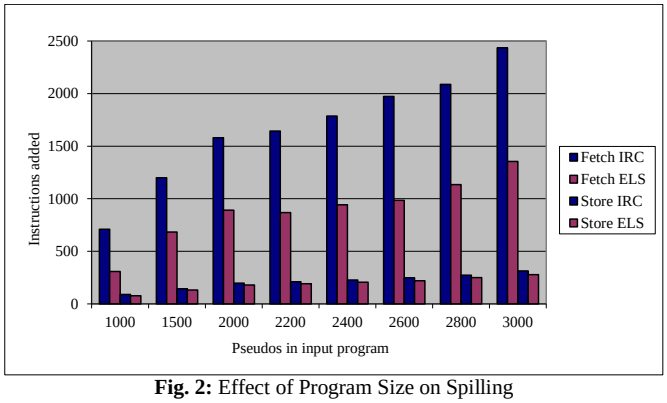

Fig. 2: Effect of Program Size on Spilling

Fig. 2 shows that the ELS_1 algorithm produces a much better allocation than IRC. Recall that the programs used are artificial and are all of a similar structure.

The fact that there is a significant difference between the two algorithms indicates that there is some merit to the idea of register-allocated paging. The operating system's (reactive) paging behavior could be viewed as another different algorithm; one that takes no account of future instructions. Given that different algorithms produce significantly different results, it is reasonable to hope that algorithms based on richer information will do better.

7.3. Comparing spilling at different register pressures

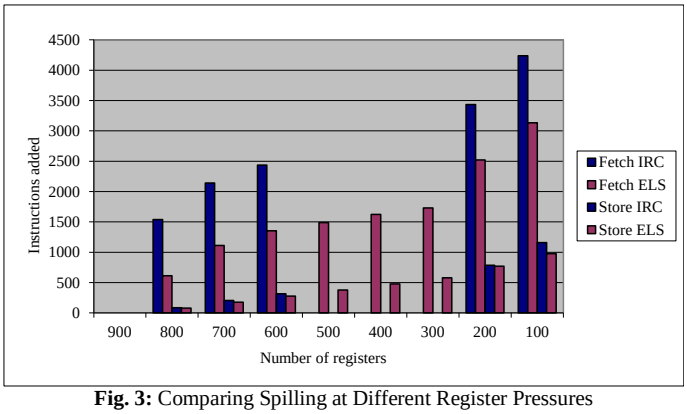

Fig. 3: Comparing Spilling at Different Register Pressures

Again we see that IRC spills significantly more than ELS_1. This is a surprising observation – the experiments conducted by Sarkar and Barik showed 5.8% as the largest runtime performance improvement achieved by ELS_1 compared to graph colouring.
In Fig. 3, no results are given for IRC in three cases. This is due to one of the design issues noted for IRC in section 6.3: it makes no distinction between spilled pseudos and those from the original program. I implemented the spill operation to fail if the pseudo to be spilled was itself the result of a previous spill, and those runs experienced this failure condition.

7.4. Resource consumption by IRC

I ran the IRC algorithm against a program containing 10,000 pseudos where a spill-free allocation required 2,874 registers. Fig. 4 shows how the run time of the algorithm varies with the register deficit (defined as 2,874 – *k*, where *k* was the number of registers).

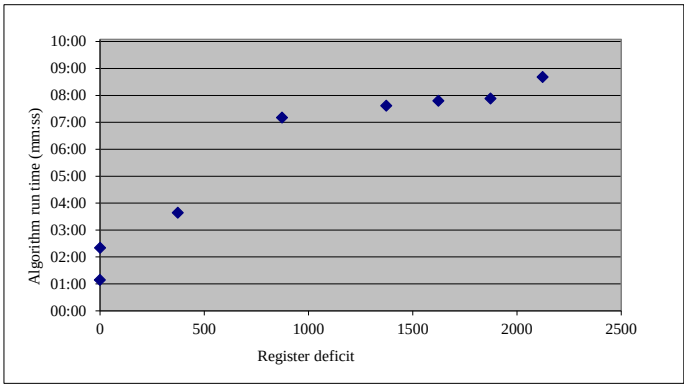

Fig. 4: IRC Algorithm Run Time with Register Pressure

It is clear from Fig. 4 that the amount of spilling (which is a consequence of the register pressure) has a dramatic effect on the run time. There are two contributing factors. Firstly, the algorithm is implemented as a loop which runs once if there is no spilling, but at least twice if spills are required (essentially, it applies its spill decisions to produce a new program, and then starts again). Secondly, in those subsequent iterations the interference graph contains many more nodes, because the effect of spilling is to replace one pseudo that has a long live range with many pseudos that have shorter live ranges.
My experience affirmed the findings of Cooper, Harvey and Torczon [14]: the choice of data structure to represent edges in the interference graph is crucial, and the best choice depends on the graph. I chose to use a bit matrix for reasons of speed, but this requires O (n^2) space, where n is the number of nodes (pseudos). The effect of spilling is to increase the number of nodes, sometimes dramatically, making the use of a bit matrix less appropriate.

7.5. Resource consumption by ELS_1

I ran the ELS_1 algorithm against a program containing 2,500 pseudos where a spill-free allocation required 722 registers (note that this program is a quarter of the size of the one used for IRC).

Fig. 5 shows how the run time of the algorithm varies with the register deficit (defined as 722 – *k*, where *k* was the number of registers). Register pressure has only a limited impact on the run time of the ELS_1 algorithm (in contrast to IRC).

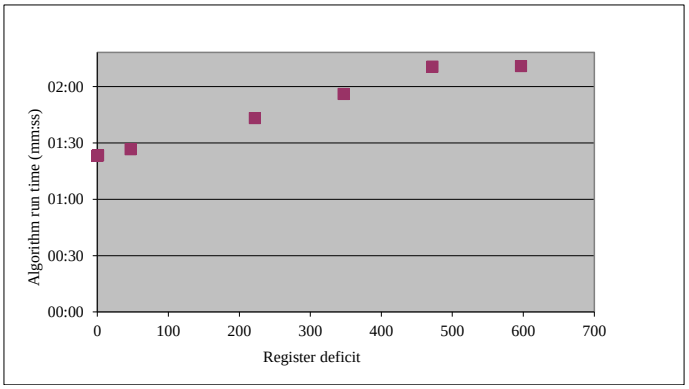

Fig.5: ELS_1 Algorithm Run Time with Register Pressure

Fig. 6 shows how the time taken to run *my implementation* of the ELS_1 algorithm varies with program size, and helps to illustrate why I did not use a 10,000 pseudo program to evaluate ELS_1 – it would simply have taken too long!

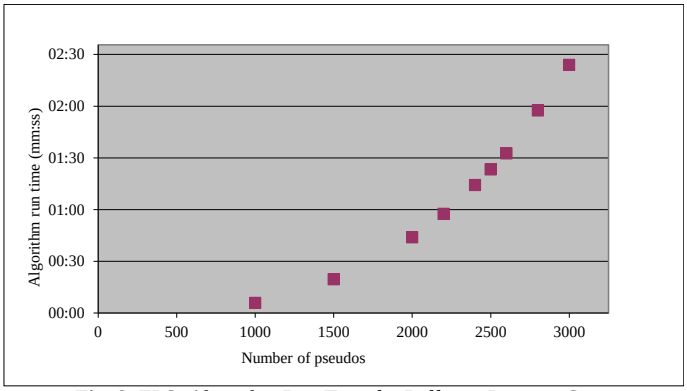

Fig. 6: ELS_1 Algorithm Run Time for Different Program Sizes

Sarkar and Barik [32] observe that "compile-time measurements depend significantly on the engineering of the algorithm implementations", and indeed the results in Fig. 6 indicate that my implementation of the algorithm does not perform as they intended. Their Theorem 4 states that the ELS_1 algorithm's time requirement is $O(progsize + intervals\,(log(count_{max}) + log(progsize))$ where *progsize* is the number of instructions in the program, *intervals* is the number of live intervals (equal to the number of pseudos in this case) and $count_{max}$ is the maximum number of pseudos live at the same time anywhere in the program. However, the pseudocode for their algorithm does not specify how to achieve this in practice: for example, step 3 of the algorithm requires "each program point *P* … in decreasing order of *freq[P]*" and within that "each symbolic register *s* in decreasing order of *totalSpillCost(s)*" but the proof of Theorem 4 states that "step 3 contribute[s] at most $O(|I|)$ time" which implies that both orderings must already have been done.

8. Conclusion

8.1. Key findings

This research set out to address the question of the applicability of register allocation techniques to a memory/disk paging problem in software which processes many large vectors. The question has two elements – is this approach feasible, and would it be worthwhile?

The design and prototype implementation go some way towards demonstrating that the approach is feasible. The principal question remaining concerns the implementation of the storage part of the vector virtual machine. There is a need to retain an index into the temporary storage file, which must either be in memory (a scarce resource) or on disk (requiring two seeks rather than one for each fetch and store).

This research also provides an experimental framework for determining the effect of a register allocation algorithm on a vector program. The experiments conducted show that register allocation algorithms can be applied to vector programs, and that the choice of algorithm has a significant impact on the quality of the allocation, and the time taken to achieve it. As expected, the experiments showed a slower algorithm producing a better allocation, though in contrast to the

expectation set by its authors, Extended Linear Scan [32] was that slower algorithm. Whilst I gave approximately equal attention to each algorithm implementation, both would benefit from profiling and tuning. The pseudocode for Extended Linear Scan does not make it clear how the claimed performance characteristics are achieved (this is part of my more general observation that this implementation required a good deal more interpolation between the lines of pseudocode than the implementation of Iterated Register Coalescing). Secondly, the description does not specify how the outcome of the algorithm is to be applied to the program, and in particular which register a spilled pseudo should be stored from or fetched into.

The experimental results are based very heavily on one family of similar programs. These programs were invented specifically for this research: they were designed to exhibit some of the features of real vector programs, and to challenge the algorithms. Even once the algorithms were tuned, it would be unwise to select an algorithm based solely on their performance with one or two programs. In particular, the experimental results presented by Sarkar and Barik [32] showed only a small improvement in estimated program run time using Extended Linear Scan rather than graph colouring, whereas my results suggested a significant improvement.

8.2. Future research

In order to show that the approach of register-allocated paging has practical benefit, a complete vector virtual machine capable of executing programs could be developed. Options for storage could be implemented and evaluated. A simple storage implementation, which does not use disk directly but maintains all vectors in virtual memory and allows the operating system to do the paging, could be used to compare register-allocated paging with a simplistic alternative.

Several questions arise from my algorithm implementations. For ELS_1: Was the poor run time a facet of my implementation and not the algorithm itself? How are the spilling decisions to be implemented? For Iterated Register Coalescing: Can the spilling performance be improved - perhaps by changing the spill cost metric? Is there a better way to store the edges of the interference graph, particularly after the program has been rewritten for the first time? How can the spilling of pseudos which resulted from a previous spill be prevented?

This research was motivated by a problem encountered by the developers of a commercial software product: Towers Watson's Igloo™. The example programs used in this research were artificially created and have little practical relevance. A compiler could be written to transform real models, written in Igloo's proprietary high-level functional language, into the register-allocated paging prototype's language. These would provide a more suitable benchmark upon which to base further experiments.

Both of the algorithms used in this research require a cost function to guide their spilling decisions. My literature search did not explore cost estimation in any detail. In addition, the characteristics of the particular storage mechanism in use (the number of spindles in the disk, for example) may affect the cost of storing and fetching. Cost estimation is not just important for making spill choices: in order to estimate the total cost of execution of the program, we need to estimate how long each instruction will take. Total execution cost is important because it provides a basis for comparing two register allocations, and for predicting actual run time.

The graph colouring algorithm used in my experiments is relatively old. The range of algorithms could be extended – in particular, a number of efficiency gains are possible for programs in Static Single Assignment form.

Acknowledgements

I would like to thank my Open University supervisor, Dr Kevin Curran, for his guidance and feedback and my employers, Towers Watson (formerly EMB), for their generous allowance of study time. I also acknowledge the support of my colleagues in the Igloo™ development team, in particular David Christensen who introduced me to the research topic.

References

[1] Duffy, D.J. and Kienitz, J. (2009) 'Monte Carlo frameworks: building customisable high performance C++ applications', Chichester, John Wiley & Sons.

[2] Tian, X. and Benkrid, K. (2008) 'Design and Implementation of a High Performance Financial Monte-Carlo Simulation Engine on an FPGA Supercomputer', *ICECE Technology International Conference on Field Programmable Technology*, pp 81–88.

[3] Nandivada, V. K. (2007) 'Advances in Register Allocation Techniques', *the Compiler Design Handbook: Optimizations and Machine Code Generation*, CRC Press.

[4] Muchnick, S. S. (1997) 'Advanced Compiler Design and Implementation', San Francisco, Morgan Kaufmann Publishers, Inc.

[5] Appel A. W. and Palsberg, J. (2002) 'Modern Compiler Implementation in Java', 2nd Edition, Cambridge University Press, Cambridge.

[6] Pereira, F. M. Q. and Palsberg, J. (2005) 'Register Allocation via Coloring of Chordal Graphs', *Lecture Notes in Computer Science*, vol. 3780/2005, pp 315–329.

[7] Chaitin, G. J. (1982) 'Register allocation and spilling via graph coloring', *Symposium on Compiler Construction*, vol. 17, no. 6, pp 98–105.

[8] Blazy, S. and Robillard, B. (2009) 'Live-Range Unsplitting for Faster Optimal Coalescing', *Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, ACM, New York.

[9] Dowd, K. and Severance, C. (1998) 'High Performance Computing', Sebastopol, O'Reilly & Associates, Inc.

[10] Li, L., Gao, L. and Xue, J. (2005) 'Memory Coloring: A Compiler Approach for Scratchpad Memory Management', *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pp 329–338.

[11] Yang, X., Deng, Y., Wang, L., Yan, X., Du, J., Zhang, Y., Wang, G. and Tang, T. (2009) 'SRF Coloring: Stream Register File Allocation via Graph Coloring', *Journal of Computer Science and Technology*, vol. 24, no. 1, pp 152–164.

[12] The Open University (2001) MT365 *Graphs, Networks and Design*, Milton Keynes, Open University.

[13] Briggs, P. (1992) 'Register Allocation via Graph Coloring', (PhD Thesis) Rice University.

[14] Cooper, K. D., Harvey, T. J. and Torczon, L. (1998) 'How to Build an Interference Graph', *Software – Practice and Experience*, vol. 28, no. 4 (April), pp 425–444.

[15] Galinier, P. and Hao, J.-K. (1999) 'Hybrid Evolutionary Algorithms for Graph Coloring', *Journal of Combinatorial Optimization*, vol. 3, pp 379–397, The Netherlands, Kluwer Academic Publishers.

[16] Glass, C. A. and Prügel-Bennett, A. (2003) 'Genetic Algorithm for Graph Coloring: Exploration of Galinier and Hao's Algorithm', *Journal of Combinatorial Optimization*, vol. 7, pp 229–236, The Netherlands, Kluwer Academic Publishers.

[17] Mahajan, A. and Ali, M.S. (2008) 'Hybrid Evolutionary Algorithm for graph coloring register allocation', *IEEE Congress on Evolutionary Computation*, Hong Kong, pp1162–1167.

[18] Cooper, K. D., Schielke, P. J. and Subramanian, D. (1999) 'Optimizing for Reduced Code Space using Genetic Algorithms', *Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers and tools for embedded systems*, New York, ACM.

[19] George, L. and Appel, A. (1996) 'Iterated register coalescing', *ACM Transactions on Programming Languages and Systems*, vol. 18, no. 3, May 1996, pp 300–324, New York, ACM.

[20] Park, J. and Moon, S.-M. (2007) 'Optimistic Register Coalescing', *ACM Transactions on Programming Languages and Systems*, vol. 26, no. 4, July 2004, pp 735–765, New York, ACM.

[21] Odaira, R., Nakaike, T., Inagaki, T., Komatsu, H. and Nakatani, T. (2010) 'Coloring-based Coalescing for Graph Coloring Register Allocation', *CGO '10 Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*.

[22] Koes, D. R. and Goldstein, S.C. (2009) 'Register Allocation Deconstructed', *12th International Workshop on Software & Compilers for Embedded Systems*, ACM, New York.

[23] Bouchez, F., Colombet, Q., Darte, A., Rastello, F. and Guillon, C. (2010) 'Parallel Copy Motion', *SCOPES'10*, June 2010, ACM.

[24] Smith, M. D., Ramsey, N. and Holloway, G. (2004) 'A Generalized Algorithm for Graph-Coloring Register Allocation', *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, vol. 39, issue 6 (May), New York, ACM.

[25] Ahn, M. and Paek, Y. (2009) 'Register Coalescing Techniques for Heterogeneous Register Architecture with Copy Sifting', *ACM Transactions on Embedded Computing Systems*, vol. 8, no. 2, January 2009, New York, ACM.

[26] Appel, A. W. and George, L. (2001) 'Optimal spilling for CISC machines with few registers', *PLDI'01*, pp243–253.

[27] Hack, S., Grund, D. and Goos, G (2006) 'Register allocation for programs in SSA form', *Lecture Notes in Computer Science*, 2006, vol. 3923/2006, pp 247–262.

[28] Barik, R., Grothoff, C., Gupta, R., Pandit, V. and Udupa, R. (2007) 'Optimal Bitwise Register Allocation Using Integer Linear Programming', *Lecture Notes in Computer Science* vol. 4382, pp 267–282, Berlin Heidelberg, Springer-Verlag.

[29] Poletto, M. and Sarkar, V. (1999) 'Linear scan register allocation', *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 6.

[30] Mössenböck, H. and Pfeiffer, M. (2002) 'Linear Scan Register Allocation in the Context of SSA Form and Register Constraints', *Lecture Notes in Computer Science*, vol. 2304/2002, pp 229–246, Berlin Heidelberg, Springer-Verlag.

[31] Wimmer, C. and Franz, M. (2010) 'Linear scan register allocation on SSA forms', *CGO '10 Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization.*

[32] Sarkar, V. and Barik, R. (2007) 'Extended Linear Scan: An Alternate Foundation for Global Register Allocation', *Compiler Construction, Lecture Notes in Computer Science*, vol. 4420, pp 141–155, Berlin Heidelberg, Springer-Verlag.

[33] Pereira, F. M. Q. and Palsberg, J. (2008) 'Register allocation by puzzle solving', *PLDI '08 Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, ACM.

[34] Pereira, F. M. Q. and Palsberg, J. (2010) 'Punctual Coalescing', *Lecture Notes in Computer Science*, vol. 6011/2010, pp 165–184, Berlin Heidelberg, Springer-Verlag.

[35] George, L. and Appel, A. (1995) 'Iterated register coalescing', Technical Report CS-TR-498-95, Princeton University [http://www.cs.princeton.edu/research/techreps/TR-498-95 accessed 31-Dec-2011].