## Research Proposal: Michael O'Boyle

# Abstract

This project aims to develop a new framework which enables compilers to adapt with and exploit the exponential growth in hardware performance. This is achieved by systematically describing the program optimisation space and by developing new approaches to explore this space in a manner independent of application or underlying architecture. Furthermore, by transcending the distinction between compile-time and run-time and by integrating currently distinct areas of program comprehension and transformation, this project aims to develop a systematic optimisation approach suitable for a diverse range of requirements from cost critical hard-wired embedded applications to super-computing grand challenge applications and just-in-time compilation of downloaded applications for reconfigurable FPGA based processors. Given this foundational work, this project aims to change the area of optimising compilation from one that is *ad hoc* and reactive to hardware evolution to one based on solid foundations that drives architectural change whereby the next generation of hardware is co-designed by compiler technology.

# Introduction

Computer systems are becoming increasingly prevalent in modern society. From web-based internet computing to embedded processors within mobile 'phones, computer technology has a significant social and economic impact on our lives. The growth in the use of computing technology is matched by a continuing demand for higher performance in all areas of computing. This demand for ever greater performance has led to an exponential growth in hardware performance and architecture evolution. Such a rapid range of architectural change has placed enormous stress on compiler technology, which maps user applications onto system hardware, such that current compiler technology can no longer keep up with architectural change. Currently, state-of-the art compilers can achieve, at best, around 10% of peak performance from a typical modern super-scalar micro-processor. The problem of compiler technology keeping pace with micro-processor evolution is compounded by the simultaneous growth in global computing, giving rise to large distributed and heterogenous systems with dynamic architecture and application software. Furthermore, the increasing use of component based software, while good software engineering practice, is based on encapsulation which reduces compiler knowledge and hence reduces attainable performance. The key point is that traditional approaches to compiler optimisations are based on static analysis and transformation which can no longer be used in a computing environment that is continual changing. What is required is an approach which evolves and adapts to applications and architectural change without sacrificing performance.

This project aims to develop new theory and practice in the area of optimising compilation so as to allow future application software to exploit current and emerging hardware. This will be achieved by developing a new unified approach to program transformation and optimisation based on adaptive compilation. This framework will allow a systematic optimisation of applications ranging from video compression software running on an embedded processor to applications running on dynamically change FPGA to grand challenge problems running on high end supercomputers and distributed networks.

For compilers to be able to adapt to changing architecture, they must be able to navigate an optimisation space and have a metric, whether it be static analysis or profile information, to determine the worth of a transformation. This knowledge must not be hardwired within the compiler, as it will be quickly be made redundant with architectural change. Similarly, techniques to find good optimisation regions must be able to use appropriate static analysis to reduce the complexity of exploring enormous search spaces, while at the same time being able to adapt dynamically to new cost metrics, introduced say by changing FPGA layouts. Before such an optimisation space can be investigated, it must be defined in a manner that allows systematic representation of program alternatives in a manner amenable to optimisation. In effect, we wish to describe the space of all useful equivalent programs and have associated with each point a measure of cost. Once we have a uniform approach to describing the program transformation space and effecient adaptive techniques for searching this space for good solutions, it can then be extended to investigate the program/ architecture space allowing software and hardware co-design. This body of work can be broadly divided into 3 themes: *Theme 1*: Defining the program optimisation space, *Theme 2* Adaptive Optimisation: Traversing the program optimisation space and *Theme 3* Co-design in the programme/ architecture space.

# Background

**Theme 1: Unifying program transformation theory/ Defining the program optimisation space**
The design space of an optimising compiler is the domain of all possible equivalent programs. In other words, given a current program, what are the possible equivalent programs which have exactly the same semantics, but

in general will differ in space and time efficiency. There have been several alternative approaches to describing such a program optimisation space including those more focussed on describing equivalence based on semantic reasoning [26] and those more interested in optimisation based on data dependence information. The most widely used approach to program optimisation is based on control-flow restructuring using data dependence information [33]. In particular, most modern compilers perform a variety of loop based transformations such as permutation, distribution and tiling to generate high quality code and recent years have seen a great improvement in loop transformation theory. By using an affine representation of loops, based on polyhedral models, several loop transformations have been incorporated into one single framework. In [2], Banerjee shows that loop interchange, reversal and skewing can be described as unimodular transformations of the iteration space. In [20], Li and Pingali extend the class of transformations to non-singular matrices, thereby incorporating loop scaling. Furthermore, this framework has been extended to statement-wise transformations [18] and integrated with loop alignment [17]. From this treatment, general validity tests have been derived. This enables compound transformations to be applied in one go rather than step by step [33]. In [14], the class of transformations is further extended and presented in a unifying framework based on schedules. Loop transformations, however, are not the only program transformations of interest when optimising programs for parallel execution. In [24, 25], I described a unifying framework for data transformations, symmetric in approach to the framework used for loop transformations. Such approaches are attractive in that it is possible to describe the ideal target form of a program based on some cost metric such as parallelism or locality which can be characterised in a linear algebraic manner. Furthermore, as the transformation space under consideration and the program representation are identical, we have direct constructive methods for finding good sequences of transformations.

**Theme 2: Adaptive Optimisation : traversing the program optimisation space** The use of transformations to improve program performance has been extensively studied for over 30 years. Such work is based primarily on static analysis and is characterised by trying (i) to determine how such a program would perform on a particular processor and (ii) developing a program transformation such that the code is more likely to execute more efficiently [30, 31]. Such an approach relies on modelling those features of the architecture and the program that are considered important. Although there has been improvement, much work remains because it is extremely difficult to accurately model program/machine interaction. The problem is made worse in that the number of transformations to apply is potentially infinite and therefore the transformation space considered must be highly restricted. Furthermore, due to architectural evolution, any optimisation strategy is swiftly made redundant. This problem is becoming more severe with each new generation of processor. Due to the problems of compile-time undecidability, several researchers have considered using runtime information to select the best implementation. They typically define one or more options statically which are then considered at runtime. For example, in [12], whether or not a portion of the iteration space should be tiled depends on runtime characteristics and in [9], different synchronisation algorithms are called depending on runtime behaviour. In [22], genetic algorithms are used to create and select transformations for parallel optimisation. In [16, 6], we examined another approach to this problem by first describing the problem as that of searching a non-linear optimisation space in order to find the minima. Analytic techniques typically consider one parameter in the optimisation space at a time to avoid dealing with combinatorial problems. In practice, however, the program transformations are not independent in their effect on performance but are in fact closely coupled. Furthermore, the optimisation space is highly non-linear with many local minima.

Recent work in just-in-time compilation for Java first interprets and then compiles for the target machine. This approach is inherently adaptable as long as a Java virtual machine is available. Currently, this work is woefully lacking in performance. Even when just-in-time compilation is used, it is invariably just too late i.e. it is on the critical part of the program execution and is therefore simplistic in order to reduce actual execution time. At present, optimisations are either based on trying to recover the original syntax in order to perform rudimentary register allocation [1], or trying to generate native machine code directly with basic invariance and redundancy elimination optimisations [4]. An alternative approach described in [10] is based on runtime translation of existing machine code for one architecture and mapping it to another. Whilst allowing portability and hence adaption to architecture, it cannot benefit from high level structural changes which as shown in [6] can be critical for high performance. In a complete different area of research, program reflection [15] can be thought of as runtime program specialisation where knowledge about data values can be used to simplify programs, generating more efficient specialised code. It has yet to be integrated with other approaches to runtime optimisation and makes no attempt at architectural specific improvement.

**Theme 3: Integrated Compiler/Architecture Design** The rapid growth in transistor densities has made it possible to gradually include more sophisticated hardware mechanisms for maintaining data flow across wider groups of parallel function units. In [27], Yale Patt *et al.* proposed that future microprocessors will attempt to issue between 16 and 32 instructions per cycle, and have upwards of 48 pipelined functional units sharing 2000 reservations stations. Prediction and speculation will be essential to maintain the high level of data flow in such processors [13, 19]. This approach however cannot be sustained as the complexity of wakeup logic and the instruction window will come to dominate the cycle time if todays architectures are scaled naively to exploit available chip area [11]. As a consequence, future micro architectures will be clustered. This may include groups of functional units, multi-threaded processors or multi-processors on a chip. Whatever the form of internal units, it means weaker communication paths between instructions and more work for the compiler, which must then partition and schedule the code to minimise inter-cluster communication. Modern VLIW processors [29] are a current example where architecture has been simplified, placing more of the responsibility on the eventual compiler. Indeed, anecdotal information concerning Intel suggests that the critical component in the launching of the merced/IA64 processor is the development of the compiler rather than hardware - thus even the world's largest processor manufacturer requires advances in compiler technology in order to utilise their products.

The issue of co-design has been long addressed in specialised embedded systems [21]. For each new application, a specialised architecture and program was developed to maximise available resources. As embedded systems increasingly resemble instruction set architecture processors (due to economy of scales associated with PC architecture), the co-design community is increasingly examining the issue of code generation as critical to their work. Thus we have trends from two directions, namely main stream micro-processors and specialised embedded systems, both requiring that compilers and hardware can work together to improve overall system performance. The question remains to what extent can compiler knowledge effectively guide hardware design and to what extent compiler strategy can adapt to hardware evolution?

# Methodology

This section describes the overall aims and objectives which is followed by a detailed description of the research programme and an outline of the workplan and associated milestones.

## Aims and Objectives

The overall aim of this project is: to develop a new framework which enables compilers to adapt with and exploit the exponential growth in hardware performance. This can be divided into a series of technical objectives:

- Unifying optimising program transformation theory.
- Formally describing program equivalence space in a manner suitable for applying a wide class of optimisation strategies.
- Express optimisation in a manner independent of source of analysis and independent or compile or runtime context.
- Investigate novel optimisation techniques including computational learning approaches.
- Extend the program design space to include architectural space and consider design trade-offs using exploitation of the combined design space.
- Apply co-design techniques to clustered processors and dynamic determination of processor configuration.
- Prototype any promising techniques and evaluate their performance relative to state-of-the art approaches.

Once these aims have been achieved this project aims to change the area of optimising compilation from one that is *ad hoc* and reactive to hardware evolution to one based on solid foundations that drives architectural change whereby the next generation of hardware, is co-designed by compiler technology.

## Programme of work

**Theme 1** The main objective of this theme is to provide a more unified descriptions of program transformations. This will allow a systematic mechanism to describe the space of equivalent programs which is amenable to static and dynamic optimisation strategies. I first intend to fully examine the properties of rank-modifying transformations developed by myself in [25]. Although extremely powerful, at present such transformations must be represented as sequences of transformation and I wish to develop a calculus defining how such sequences may be manipulated. This will be shortly followed by an investigation of direct constructive generation of optimisation transformations in a manner similar to the non-singular case [24]. Although it appears, in many *ad hoc* examples that direct construction is possible, it has yet to be proved. This work will aim to go beyond the expressive powers of the time/space scheduling of systolic synthesis allowing straightforward optimisation criteria. Once this highly technical work has started, I wish to investigate the use of a limited set of transformation function that may be combined with integer algebraic transforms to describe the class of all known loop and data transformations. The challenge here is whether a formulation can be designed in such a manner that is complete without losing the structure ( and hence power ) of existing models. This will lead to a hierarchical classification of program types from linear, piece-wise linear, parametric rank-modifying up to monotonic and beyond where each class will have an associated program transformation representation.

Following on from this unification work, I would like to consider the issue of mapping. Mapping occurs when one goes from a sequential to parallel program and from a sequential node program to compiler scheduled VLIW code. Although we can reason about sequences of transformations, mapping inserts an artificial barrier. For instance, it is difficult to reason about the impact of pre and post partitioning transformations leading to the well-known phase-order problem. Similarly, source to source transformations and their impact on eventual scheduled machine code are little understood. I believe that the work described in [25] can be extended so that parallelisation and hence work and data mapping can be expressed in the same framework as standard program transformations, allowing reasoning across the transformation phase-order.

The next step would be extending the control structure under consideration to recursion and its associated data structures including trees, In [7], Jean-Francois Collard integrates recursive analysis with linear access and static control-flow. His work implicitly uses a new set of (undefined) transformation. I hope to explicitly determine such a set of transformation and formulate them in a manner suitable for direct construction from cost-based analysis. Continuing along this line of investigation leads to determining the class of transformations relevant to commutative analysis which has be shown to be useful in object-orientated optimisation. Although highly ambitious this will be of direct benefit in the development of JIT compilers for object orientated languages such as Java. Overarching all this work is the need to express what in fact is our design space in such a manner that it can be usefully explored.

**Theme 2** The key objective of this theme is to develop anytime optimisation techniques that adapt to changing circumstance i.e. optimisation is invariant of compile-time or runtime context. By building a compiler framework that allows seamless movements across context, we can exploit and adapt to the currently available resources. Current approaches to runtime optimisation are *ad hoc* in approach and fundamentally severely restricted in scope. One of the most technically advanced projects, Jalapeno [4], applies classical dataflow transformations on assembler if a particular routine is frequently used. The best that this approach can ever achieve is the same performance as static optimisation if the code is known at design time. In contrast, once the basic anytime framework is in place, the next step is to investigate the optimisation space of interest described in theme 1. Thus given some information whether it be static analysis or profile information gained on-line/off-line which serves as a goodness metric, can we find improved programs in a manner than can cope with evolving hardware? Initial work will look at generic techniques such as genetic algorithms or classical non-linear optimisations such as simulated annealing or conjugant gradient, to search the optimisation space for minima. A key consideration is the extent to which it is possible to modularise any optimisation strategy so that knowledge about the program/architecture is minimised. If the space changes due to underlying system modification, we wish such techniques to be still applicable. Later work will look at architectural neutral approaches to discovering characteristics of the transformation space drawing on ideas from computational learning [3] where test programs are generated as "questions", executed, and the execution time forming "answers" which can be used to calibrate the hardware.

The fundamental limitation of such an approach are that the program equivalence space is infinite and in sampling a small section, the overhead of evaluation must be considered.. The next step is therefore is to add budget models to the compilation strategy where different optimisation techniques will be applicable depending on budget. Critical to determining whether it is profitable to pursue optimisation will be analytic upper and lower bound on achievable performance which can be used as termination criteria. These dynamic approaches whist attractive in that they directly determine execution cost are inevitably expensive and in all but the most trivial case will require static analysis to prune the decision space forming optimisation hyperplanes. The key research issue, is to incorporate static analysis in such a manner that it reduces optimisation complexity without compromising the compiler's ability to adapt.

**Theme 3** The overall objective of this theme is to extend the ability of compilers to adapt to hardware evolution and then influence such evolution. Initially, I will consider a limited application domain namely, multimedia applications which form the main task of so-called media processors. Such applications have relatively data independent control-flow and certain program behaviour dominates providing an ideal restricted domain with which to experiment. Initial work will consider a parameterised VLIW architectural space, similar in scope to Trimaran [32], with a fixed compilation strategy. Rather than searching through the program space at design time, we will now search a highly limited architectural space. If we wish to examine embedded systems it may be appropriate to consider just one application program, for more general purpose processors, many benchmarks must be selected and a judgement made. The next step would be to allow both the compiler to traverse the program design space, and for the architecture space to also be traversed - providing genuine co-design, albeit within a restricted architectural domain. Following on from this, I would then consider clustered architectures. This will require fundamental work in theme 1 to describe mapping (partitioning/scheduling) and other transformations in one framework so that inter and intra cluster scheduling can be considered symmetrically.

Processors with multiple internal clusters will inevitably have to consider internal memory coherence as access times across chip start increasing beyond 20 cycles, so one obvious avenue of work is investigating whether my previous work in coherence across multiprocessors [23] can be translated into a uni-processor, multi-cluster setting. In order to achieve this navigation of the program and architecture design space, we need to parameterise both the machine model and code generator in a manner that describes the underlying instruction set's syntax, semantics and cost [28]. The research challenge here is to find a description for which it is possible to automatically translate and optimise high-level programs into machine code and simultaneously be able to describe a large enough set of interesting architectures. Another issue is whether there are there symmetric techniques to describe the program and architecture space? Finally, these techniques which have been aimed at design time evaluation of program/processor combinations can now be extended to execution -time. As programs execute, we can monitor the compiler's ability to exploit machine resources, and if necessary, in the case of FPGAs, adapt the underlying hardware to fit the current situation. Finally, as FPGAs seem likely to continue to be less efficient than hardwired implementations, program and processor adaption can be investigated in a hybrid architecture with a RISC core and FPGA as the surrounding real estate [5].

## Workplan

Any workplan spread over 5 years is likely to be approximate especially with regard to the more speculative parts of the proposal. It is my intention to review progress in the light of current development at the end of each year and update the workplan accordingly.

**Theme 1**

- Extend iteration and index space representation to cover the widest form of transformation space. (1.1)
- Develop optimisation oriented framework on recursive calls and tree data structures. (1.2)
- Investigate use of commutativity analysis to unify object, array and recursive structures. (1.3)
- Look at role of architecture indifferent optimisations such as abstract interpretation and program specialisation. (1.4)

**Theme 2**

- Formulation of optimisation as an anytime activity. (2.1)
- Investigate the application of novel search techniques based on the formulation of the program optimisation space. (2.2).
- Investigate auto-generation of test cases for computational learning approaches so as to characterise the space. (2.3)
- Investigate use of static analysis/ constructive transformations as forming search hyperplanes in space. (2.4)

**Theme 3**

- Investigate a small VLIW parameterised space. (3.1)
- Develop a machine model suitable for describing semantics and cost of a machine model. (3.2)
- Iterative co-design experiments in a limited architectural domain. (3.3)
- Application of design techniques to clustered and dynamic architectures. (3.4)

**Milestones** There are five milestones roughly corresponding to one a year- which can be seen on the workplan chart. Each of the sub-tasks (1.1) through to (3.4) has a time-line associated with it, and at each milestone a number of tasks should be completed and a written report available. At Milestone 1: (12 months) 1.1 and 2.1 will be complete; at Milestone 2: (24 months) 1.2, 2.2 and 3.1; at Milestone 3: (36 months) 1.3, 2.3 and 3.2; at Milestone 4: (52 months) 1.4, 2.4 and 3.3; and at Milestone 5: (60 months) 3.4.

# References


[1] Azvedo A. Nicolau and Hummel J., "Java Annotation Aware: Just-In-Time (AJIT) Compilation System" ACM Java Grande Conference,1999.

[2] Banerjee, "Unimodular Transformations of Double Loops", 3rd Workshop on Programming Languages and Compilers for Parallel Computing, 1990.

[3] Betke M., Rivest R. and Singh M., "Piecemeal Learning of an Unknown Environment" Machine Learning 28:2-3,1995.

[4] Burke M et al, "The Jalapeno Dynamic Optimizing Compiler for Java" ACM Java Grande Conference, 1999.

[5] Donlin A., "Self Modifying Circuitry - A Platform for Tractable Virtual Circuitry", International Workshop on Field-Programmable Logic and Applications, LNCS, 1998.

[6] Bodin F., Kisuki T., Knijnenburg P.M.W., O'Boyle M.F.P., and Rohou E. "Iterative compilation in a non-linear optimisation space", Proc. Workshop on Profile and Feedback Directed Compilation,1998

[7] Cohen A., Collard Jean-Francois, Djelic I. and Feautrier P., " A Transduction-based Framework for Recursive Program Analysis Programs with Static Control", Tech Rep, U. Versailles, 1998

[8] Colwell, R., "Challenges and Trends in Processor Design", IEEE Computer, 31(1), January 1998.

[9] Diniz P. and Rinard M., "Dynamic Feedback: An Effective Technique for Adaptive Computing", PLDI,1997.

[10] Ebcioglu K. and Altman E.R., "DAISY: Dynaimic Compilation for 100 % Architecture Compatability", TRRC 20538, IBM Research, 1996.

[11] Fernandes M., Topham N. and Llosa J., "Partitioned Schedules for Clustered VLIW Architectures", in Int. Parallel Processing Symp.,1998.

[12] Hummel S.F, Banicesu I,, Wang C.T. and Wein J. "Load balancing and data locality via fractiling: An experimental study", Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers, 1996.

[13] Jacobson Q., *et al.*, "Control Flow Speculation in Multiscalar Processors", HPCA IEEE Press, 1997.

[14] Kelly W.A. and Pugh W., "A Framework for Unifying Reordering Transformations", CS-TR-3193, U. Maryland, 1993.

[15] Kirby G., Mossison R. and Stemple D, "Linguistic Reflection in Java", Software Practice and Experience, 28:10,1998.

[16] Kisuki T., Knijnenburg P.M.W. , O'Boyle M.F.P., Bodin F. and Wijshoff H.A.G. "A Feasibility Study in Iterative Compilation", the International Symposium on High Performance Computing, 1999.

[17] Kulkarni D. and Stumm M.,, "CDA Loop Transformations", Languages, Compilers and Run-Time Systems for Scalable Computers, 1995.

[18] Knijnenburg P.M.W., Ayguadé E. and Torres J., "Multi-transformations of Nested Loops for Parallelizing Compilers", TR 96-14, Leiden U. ,1996.

[19] Lipasti M.H. and Shen J.P., "Exceeding the Data-Flow Limit Via Value Prediction", IEEE Micro., 1996.

[20] Li W. and Pingali K., "Access Normalisation: Loop Restructuring for NUMA Compilers", Cornell University, Dept CS, TR92-1278 (1992).

[21] Marwedel P. and Goossens G., "Code generation for embedded processors", Kluwer 1995.

[22] Nisbet A. "GAPS: Genetic algorithm optimised parallelization" Workshop on Profile and Feedback Directed Compilation, 1998.

[23] O'Boyle, M.F.P., Ford, R.W., Nisbet, A.P., "A Compiler Algorithm to Reduce Invalidation Latency in Virtual Shared Memory Systems", IEEE PACT'96, October 1996.

[24] O'Boyle M.F.P and Knijnenberg P.M.W., "Non-Singular Data Transformations: Definition, Validity and Application", ICS,July 1997.

[25] O'Boyle M.F.P and Knijnenberg P.M.W., "Integrating Loop and Data Transformat ions for Global Optimisation" PACT '98, , IEEE, October 1998.

[26] Partsch H.A., "Specification and Transformation of Programs", Springer-Verlag, 1990.

[27] Patt, Y.N., *et al.*, "One Billion Transistors, One Uniprocessor, One Chip", *IEEE Computer* 30(9), 1997.

[28] Rohou E., Bodin F. and Seznec A., "SALTO: System for Assembly-Language Transformations and Optimisation", Workshop on Compilers for Parallel Computers, Germany, 1996.

[29] Sakellariou R., Stöhr E.A. , O'Boyle M.F.P, "Compiling Multimedia Applications on a VLIW Architecture", DSP97, Santorini, July 1997.

[30] Temam O., Granston E.D. and Jalby W. "To Copy or Not to Copy: A Compile-Time Technique for Assessing When Data Copying Should be Used To Eliminate Cache Conflicts", Supercomputing '93 November 1993.

[31] Temam O., Ficker C. and Jalby W., "Cache Awareness in Blockinf Techniques", TR 1998/24 U. Versailles, 1998.

[32] http://www.trimaran.org

[33] Wolf M.E. and Lam M.,"A Data Locality Optimizing Algorithm", Proceedings of Programming Language Design and Implementation, Toronto, ACM P ress, June 1991.

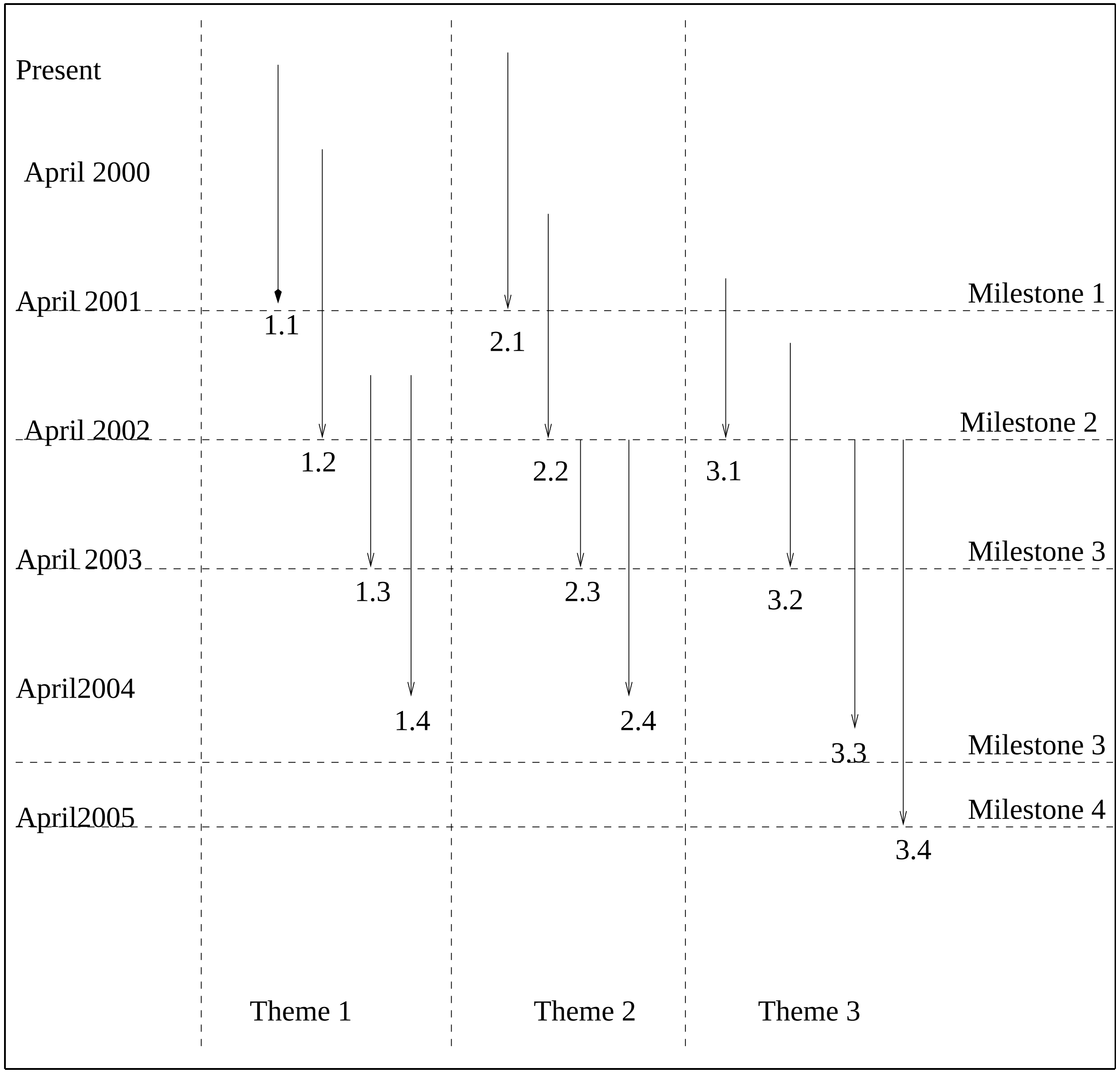


Figure 1: Workplan of proposed research